DEPARTAMENTO DE LENGUAJES Y SISTEMAS INFORMÁTICOS E INGENIERÍA DEL SOFTWARE

Facultad de Informática
Universidad Politécnica de Madrid

TRABAJO FIN DE CARRERA

SISTEMA DE MONITORIZACIÓN MULTIPROTOCOLO PARA REDES DE NODOS AVANZADOS

Autor
Carlos Antonio Bazaga Ortiz de la Tabla

Tutor
Genoveva López Gómez
Cotutor
David Lizcano Casas

Año: 2013

Agradecimientos

Con la realización de este proyecto de Fin de Carrera concluye mi etapa como estudiante y comienza la vida laboral. Han pasado muchos años desde que comencé mis estudios en esta carrera que, con estas líneas, llega a su final. Durante este tiempo no todo ha sido fácil pero la idea de llegar a este punto me ha mantenido firme y sin abandonar.

Deseo agradecer todo su apoyo a todas las personas que han ido formando parte de mi vida durante toda esta etapa y que han contribuido en ella. Todas esas personas que sin su presencia y apoyo no habría sido posible alcanzar este hito en mi vida. Agradecer a mi familia la posibilidad de realizar estos estudios durante todo este tiempo, su paciencia, su apoyo, su aguante y su comprensión.

A mis amigos por haber estado ahí durante todos estos años, sin ellos no habría sido capaz de aguantar las presiones y tensiones de los momentos más duros. Cada una de esas ocasiones ha aportado, de una forma u otra, un pequeño empujón para alcanzar esta meta.

Deseo agradecer a mis compañeros de universidad, a los miembros de las asociaciones especialmente ASCFI y CITFI, a todas las personas de la Delegación de Alumnos y todos aquellos que de una forma u otra han compartido mi tiempo durante estos años en la universidad, el haberme permitido compartir con ellos todos estos años de carrera, todas las actividades que hemos realizado juntos y los proyectos que hemos llevado a cabo y también, por qué no, las practicas que hemos compartido.

Agradecer también al profesorado de la facultad que ha estado ahí para enseñarme y ayudarme en todo lo que han podido. A aquellos que se han esforzado por enseñarnos y apoyarnos y sobre todo a aquellos que se nota que disfrutan de su trabajo haciéndolo, pues eso se convierte en un incentivo al estudiante cuando lo percibe y consigue que disfrute también de su aprendizaje.

En especial agradecer a todos los miembros del laboratorio de redes y en concreto a Genoveva por darme la oportunidad de trabajar y realizar este proyecto en un entorno laboral agradable y en contacto con el mundo profesional. A mis compañeros directos de proyecto, Alfredo y Alfonso, por ponerme en contacto con el laboratorio y por compartir las horas y esfuerzos del trabajo. También a mis tutores, Genoveva y David, por su labor en este proyecto, que sin su guía no habría llegado a ser lo que es.

Carlos Bazaga Ortiz de la Tabla

Índice General

Pág.

Pág.

Índice de figuras

Pág.

Pág.

Pág.

Capítulo 4. GEMA

Capítulo 5. Caso de uso

Capítulo 1

Introducción y Objetivos

Índice

En los últimos años las redes de comunicaciones han crecido enormemente tanto en tamaño como en variedad de soluciones tecnológicas. Estas soluciones raramente se implementan de una forma homogénea pudiendo encontrarse variedad de ellas en las grandes arquitecturas de redes de las empresas actuales, redes que se extienden en un amplio espacio geográfico. Este crecimiento ha ido acompañado de un incremento en la demanda de calidad y fiabilidad de las comunicaciones.

Una de las tecnologías que más ha avanzado permitiendo cubrir estas demandas son las redes de fibra óptica, una tecnología que permite interconectar grandes distancias sin necesidad de repetidores. Este medio de comunicación otorga un gran ancho de banda y su instalación es sencilla, pero tiene un problema: las largas distancias que cubre, del orden de decenas de kilómetros, están sujetas a averías e incidencias ante las cuales es necesario disponer de un adecuado mecanismo de monitorización.

Parejo al avance de esta tecnología han evolucionado los dispositivos para manejarla, nodos cada vez más complejos dotados de mayor capacidad y funcionalidad. Con ellos es posible diseñar enormes y complejas redes de computadores, sin embargo estos equipos, redes, su diseño y mantenimiento tienen un coste elevado, lo que hace necesaria la aparición de buenas herramientas de administración y monitorización remota capaces de gestionarlos.

Para administrar estas redes, cada vez más heterogéneas y complejas, se precisa de herramientas automatizadas de gestión y administración de redes avanzadas. Estas herramientas han de trabajar con diferentes tipos de hardware y ante muy diversas situaciones y eventualidades que puede acontecer en la red y por ello se hacen necesarios mecanismos estandarizados para el intercambio de información.

A raíz de estas necesidades aparece un estándar para los sistemas de gestión de redes, SNMP, que desde su aparición a finales de los años ochenta ha sufrido diferentes mejoras plasmadas en las tres versiones en uso actualmente, SNMPv1, SNMPv2 y SNMPv3.

Es en este escenario en el que nace este proyecto de construcción de un sistema de monitorización multiprotocolo para redes de nodos avanzados. Colaboración entre el CoNWeT Lab (Laboratorio de Redes de Computadores y Tecnologías de Red) de la Facultad de Informática de la Universidad Politécnica de Madrid y la empresa Fibernet, filial del grupo Teldat, especializada en el desarrollo de sistemas de comunicaciones con fibra óptica.

Este sistema, GEMA (Gestor Multinodo Avanzado), permitirá la gestión y monitorización automática de las redes y componentes de fibra óptica desarrollados por Fibernet pero sin dejar de ser capaz de cubrir la amplia variedad de soluciones existentes o futuras. Debiendo ser fácilmente ampliable y actualizable a nuevas tecnologías y protocolos que aparezcan en el escenario de las redes de comunicaciones.

1.1. Objetivos

El objetivo del proyecto es la realización de una aplicación programable para el diseño, monitorización y gestión de redes de fibra óptica con independencia del interfaz al usuario que se desee proveer.

Esta aplicación, GEMA, deberá ser altamente modular, escalable y ofrecer un API sobre el que implementar diferentes interfaces de cara al usuario o módulos externos según se requieran. Para dotar de mayor flexibilidad a estos interfaces el API se ofrecerá paralelamente como librería de programación y mediante tecnología web, permitiendo así que los interfaces cliente puedan distribuirse por la red.

La modularidad y escalabilidad de la aplicación serán, en todo momento, las prioridades en el diseño de la misma. Por lo cual se decide utilizar la orientación a objetos como paradigma de programación al considerarlo el más adecuado para cumplir con estas premisas. Para la implementación se decide utilizar Python, lenguaje interpretado orientado a objetos multiplataforma que cada vez está adquiriendo más presencia en el mercado del software.

De este modo se presentan los siguientes objetivos para el proyecto:

- La aplicación GEMA debe ser capaz de obtener de la red, mediante los protocolos correspondientes, el estado de los distintos elementos que conformen el diseño al igual que comunicarse con ellos para darles las órdenes correspondientes.
- Debe permitir definir desde cero un diseño de red completo incluyendo diferentes tipos de nodo, conexiones, relaciones y jerarquías entre ellos, así como crear ficheros de salvado de estos diseños para su posterior carga en el sistema.
- Debe permitir definir cada nodo con independencia de la tecnología de comunicación con el mismo para lo que se debe poder extender el conjunto de protocolos soportado, que incluirá de partida a SNMP en sus versiones 1 y 2.
- Podrán definirse qué elementos y características o variables de los mismos se deberán monitorizar, así como abstracciones de los mismos y comportamientos o acciones a tomar en función de sus valores. Para esto último se debe dotar al sistema de un lenguaje de scripting con capacidad de acceso y proceso suficiente para permitir al sistema tomar decisiones en función del estado de la red.

- El sistema debe permitir la conexión de elementos externos mediante programación contra el API o API web permitiendo acceso tanto al estado de monitorización y gestión del sistema como a su completa configuración y diseño de red. Esta comunicación debe poder realizarse tanto de forma síncrona mediante peticiones como de forma asíncrona mediante la subscripción a notificaciones de diferentes elementos como Traps o Logs.

1.2. Descripción general del trabajo

Para cumplir con los requisitos anteriormente planteados bajo un principio de modularidad y escalabilidad la aplicación de divide en cuatro grandes secciones o capas que se describen a continuación.

1.2.1. Gestores de comunicación en red

En esta capa se incluyen los elementos necesarios para la gestión del nivel de comunicación con los diversos dispositivos de red a monitorizar. En ella se encuentran implementados los diferentes protocolos a utilizar, inicialmente SNMP versiones 1 y 2. Constará así mismo con un gestor de protocolos de forma que se puedan implementar y añadir nuevos protocolos al sistema de forma transparente con la simple condición de que estos cumplan unos requisitos en su interfaz con el sistema.

De esta forma se podrá abstraer completamente el protocolo del funcionamiento del resto de la aplicación permitiendo su reemplazo y añadido sin provocar cambio alguno en otro módulo, quedando por ello definidos dos conceptos en esta capa.

Protocolo: Representa un protocolo específico y contiene la implementación de los mecanismos necesarios para su utilización en la comunicación con la red real. Implementan un interfaz con el sistema predefinido de forma que se unifica el comportamiento de los mismos permitiendo al sistema trabajar con independencia de su implementación.

Gestor de Protocolos: Representa al gestor de protocolos, es el encargado de dar de alta los diferentes protocolos en el sistema así como controlar el acceso a los mismos.

1.2.2. Representación y monitorización

En esta capa se implementan los elementos encargados de representar, de forma abstracta, las redes diseñadas junto con todos sus elementos y será la encargada de gestionar todas las tareas de monitorización y gestión asignadas al sistema.

Esta capa será el núcleo del sistema que, haciendo uso de las demás, conformará la aplicación GEMA al completo, se definen para ello los siguientes conceptos.

Layout: Representa el conjunto de nodos y conexiones del sistema así como sus jerarquías y relaciones que definen las diferentes redes a monitorizar, será por tanto la cima de toda la jerarquía de clases que conformarán el modelo.

Nodo: Representa a cualquier elemento de la red monitorizable de forma independiente, ya sea un elemento real como un multiplexor, un subcomponente o una abstracción conceptual de un conjunto o subdivisión de ellos.

Estará intrínsecamente relacionado con una IP sobre la que realizar tanto la monitorización como la administración de sus variables y administrará las propiedades que se definan sobre el mismo.

Conexión: Representa los elementos de conexionado de la red que no son monitorizables y que comunican en sus extremos a dos nodos. Al carecer de IP propia no contendrá variables, si contendrá propiedades que se definan de forma análoga a los nodos.

Variable: Representa un valor monitorizable y administrable en un nodo. Contendrá una referencia a un protocolo y un Conn_Data, una caja negra administrada únicamente por el protocolo, para gestionar y controlar la comunicación de red con el nodo físico, elemento real conectado en la red. De esta forma se consiguen abstraer los protocolos y su implementación interna del resto del sistema.

Propiedad: Las propiedades son elementos polivalentes dentro de un nodo o conexión. Internamente consisten en una función que devolverá el resultado de evaluar una serie de variables, propiedades o atributos del sistema definidos en su código. A un nivel más alto podrán representar abstracciones de variables, agregaciones, o incluso, comportamientos del sistema frente al estado de la red si son síncronas o en respuesta a eventos si son de tipo asíncrono.

1.2.3. Módulos internos

Esta capa consiste en una serie de módulos internos intercambiables que realizan diferentes tareas en paralelo al trabajo principal del sistema. Estos módulos deberán ser activados explícitamente dentro del código principal de GEMA. De esta forma se evita que módulos cuya labor podría comprometer la integridad del sistema sean añadidos como plugins externos simplemente conectándose al API. A pesar de ello mantiene una enorme independencia de código entre ellos y el núcleo, debiendo definirse únicamente unos interfaces mínimos a respetar. Inicialmente estos módulos son los que se enumeran a continuación.

Persistencia: Es el módulo encargado de dotar de persistencia al sistema, recibe un resumen del estado del sistema y lo guardará en una estructura serializada en disco para, posteriormente, poder recuperar esta estructura y reconstruirla en el sistema conservando todos los elementos descritos en el apartado anterior. El diseño de este módulo debe permitir dotar a estos ficheros de controles de seguridad y autenticidad y por ello se implementa como módulo interno.

Parser: Es el encargado de interpretar y compilar a código Python el lenguaje de scripting, GEMA-Script, definido para la implementación de propiedades. Su labor consiste únicamente en verificar la corrección del código siendo responsabilidad del sistema verificar si las diferentes referencias utilizadas son correctas con el layout en curso. Su implementación como módulo interno es para evitar posibles intentos de inyección de código malintencionado dentro de las propiedades del sistema.

Logger: Es el encargado de mantener un servicio de escucha en un puerto definido a la espera de la llegada de logs de los diferentes elementos de red.

Una vez recibidos estos son notificados al dispatcher para que los distribuya entre los diferentes subscriptores y nodos correspondientes.

Se implementa como módulo interno para controlar la apertura de puertos y, principalmente, por razones conceptuales al considerarse un componente del propio sistema y no un plugin que otorga una capa de abstracción superior como una GUI o funcionalidad de alto nivel por encima del sistema, como podría ser un subscriptor a los logs que genera estadísticas de errores dentro de la red.

Trapper: Módulo totalmente equivalente al Logger pero encargado de capturar y notificar los Traps de los elementos de la red.

1.2.4. API y módulos externos

Finalmente se tiene una capa de módulos externos o plugins conectados al sistema mediante el API. Serán, entre otras labores, los encargados de establecer una comunicación e interacción entre el sistema y el exterior implementando los interfaces con el usuario, GUI, o incorporando funcionalidades nuevas de alto nivel.

Esta funcionalidades podrán ser, por ejemplo, la generación de estadísticas y la presentación de las mismas en diferentes formatos al usuario. Inicialmente el proyecto consta de los siguientes componentes en esta capa.

API: Es el interfaz de comunicación con los diferentes componentes software añadidos a GEMA. Se implementa como una librería que otorga un objeto dotado de los métodos necesarios para que un componente externo pueda interactuar con el sistema. Mantendrá en todo momento la ocultación de la implementación interna así como cierto nivel de seguridad para evitar el acceso directo a los elementos privados del sistema.

API Web: En si mismo este componente es ya un plugin en GEMA, conectado por un extremo al API, actúa como intermediario neutro entre el sistema y la web. Ofrece un servicio por el que, a través de internet, diferentes software puedan solicitar operaciones sobre GEMA como si estuviesen accediendo directamente mediante el API.

Sobre este elemento se podrán implementar los controles de seguridad y autenticación que se consideren necesarios y, al ser un módulo externo, puede ser reimplementado, reemplazado o duplicado sin que ello afecte en lo más mínimo al funcionamiento, implementación o diseño de GEMA.

Menú: Es el menú principal que se mostrará en la consola al iniciar la ejecución del sistema. Permite el acceso a la administración de todo el sistema con todas sus funcionalidades desde la propia consola. A pesar de estar implementado como un módulo externo se considera un componente intrínseco de GEMA al ser el principal frontal de control del mismo.

Desde este menú se podrán activar o desactivar e incluso administrar el funcionamiento de los demás módulos externos. Al estar implementado en esta capa mediante el API será posible reprogramarlo o substituirlo por otro si se considerase necesario en un futuro, pero siempre habrá de estar presente y ser único.

1.3. Estructura del documento

Tras este capítulo de introducción se presentan, en una serie de capítulos, las diferentes partes que componen el proyecto:

En primer lugar se expondrá el estado de la cuestión, donde se presentará un análisis del protocolo de monitorización de red SNMP, principal sistema de comunicación en esta fase del proyecto.

A continuación se expondrán algunas de las diferentes aplicaciones de monitorización actualmente en el mercado, tanto de software libre como propietario, que han sido estudiadas. Se expondrán también diferentes sistemas de scripting para sistemas de monitorización existentes en la actualidad. Hablaremos de las alternativas de persistencia planteadas y sobre el paradigma de orientación a objetos y su adecuación al proyecto.

En el siguiente capítulo se detallará el diseño e implementación desarrollados para llevar a cabo los objetivos propuestos. Se presentará el diseño escogido y se justificará el mismo. Se documentarán los diferentes módulos y presentarán las estrategias escogidas para las diferentes capas, entrando en el detalle de las diferentes decisiones que han sido tomadas durante todo el desarrollo. Decisiones tanto desde un punto de vista técnico como desde uno práctico ya que no podemos obviar que este proyecto se enmarca dentro de un desarrollo conjunto con una empresa externa, Teldat-Fibernet, que tiene unas exigencias en cuanto a plazos e hitos a cumplir.

Posteriormente se presentará la aplicación GEMA finalizada en su primera fase, la que abarca este proyecto. Se continuará con una serie de casos de uso y se expondrán algunas de las ventajas y fortalezas de esta aplicación para terminar con una serie de conclusiones al proyecto y propuestas de evolución futura.

Tras estos capítulos se podrán localizar los anexos de bibliografía, glosario y documentación de diseño del proyecto.

Capítulo 2

Estado de la cuestión

Índice

En este capítulo se describe el estado actual de los sistemas de monitorización de redes y las tecnologías involucradas así como las estrategias de diseño y desarrollo planteadas para el proyecto.

Se analizarán en primer lugar los principales protocolos de monitorización de redes, SNMP en sus diferentes versiones. Posteriormente se mostrarán algunas de las soluciones de monitorización existentes, tanto comerciales como libres, y sistemas de scripting para la automatización de las labores de gestión. Finalmente se plantearan diferentes alternativas de persistencia planteadas y la adecuación del paradigma de orientación a objetos al proyecto.

2.1. Protocolos de monitorización de red

Existen diferentes protocolos de monitorización de red en la actualidad: SNMP (Simple Network Management Protocol) [1], SGMP (Simple Gateway Monitoring Protocol) [2] reemplazado ya por SNMP, RMON (Remote Network MONitoring) [3] que fija definiciones sobre el uso de SNMP, CMIP (Common Management Information Protocol) [4] protocolo en competencia con SNMP pero, dada su complejidad, menos extendido en internet, su uso es más habitual en sistemas de telecomunicaciones. De todos ellos el estándar aceptado e implementado por la mayoría de fabricantes y operadores de redes es SNMP, incluido Fibernet, quedando fijado así como un requisito es el protocolo en el que se centra el estudio.

SNMP presenta en la actualidad 3 diferentes versiones que estudiaremos a continuación. Estas versiones han ido añadiendo diferentes mejoras en las áreas de rendimiento, comunicación y seguridad del protocolo aunque algunas de estas mejoras no fueron aceptadas por la mayoría de los fabricantes y aparecieron versiones alternativas o reducidas del protocolo hasta la aceptación final de la versión 3 como estándar completo por el IETF (Internet Engineering Task Force) [5] en 2004.

El protocolo SNMP [6], [7], [8] se compone de una serie de especificaciones para administración de redes incluidas definiciones de estructuras de datos, sus principales elementos son tres: dispositivos, agentes y sistemas de administración de red.

- Dispositivo: Es un dispositivo de cualquier índole (routers, servidores de acceso, switchs, PCs, impresoras, etc.) conectado a la red administrada el cual cuenta con un agente SNMP y dispone de una interfaz que permite comunicación unidireccional o bidireccional con el agente. Estos dispositivos recogen y almacenan información de administración que queda disponible a los sistemas de administración por medio de sus agentes.
- Agente: Es un módulo de gestión situado en un dispositivo de la red, el agente dispone de la información de administración del dispositivo y se comunica con el sistema de administración por medio del protocolo SNMP tanto de forma pasiva al atender las peticiones de lectura o escritura o de forma activa al comunicar un evento del dispositivo por medio de los Traps. Toda esta información de administración se organiza en forma de variables que quedan definidas en una base de información de administración MIB (Management Information Base).
- Sistema de administración: Consiste en una aplicación o conjunto de aplicaciones que monitoriza y controla los dispositivos de una red, accediendo a la información y gestionándolos mediante el protocolo SNMP y reaccionando o comunicando esta información a los administradores del sistema.

La comunicación entre los agentes y los sistemas de administración se realiza mediante cuatro tipos de comando SNMP:

- Lectura: Es el más utilizado de los cuatro, solicitado por el sistema administrador a los diferentes agentes para obtener el estado de las variables en los dispositivos, es el principal medio de monitorización ofrecido por SNMP.
- Escritura: Solicitado por los sistemas de administración para cambiar el estado de las variables de los dispositivos a fin de controlar y gestionar su comportamiento.
- Trap: En este comando la iniciativa de la comunicación la toma el agente comunicando a los sistemas de administración la ocurrencia de un evento concreto en el dispositivo para su tratamiento inmediato de forma asíncrona.
- Transversales: Operaciones realizadas por el sistema administrador para determinar las variables soportadas por un dispositivo e información relacionada.

Por definición SNMP utiliza la notación sintáctica abstracta ASN1 (Abstract Syntax Notation One) [9] para estandarizar el formato de comunicación entre diferentes sistemas y dispositivos.

2.1.1. Base de información de administración SNMP (MIB)

Una base de información de administración, MIB, es una colección de información organizada jerárquicamente en forma de árbol que describe y representa las variables de un dispositivo.

Cada objeto en la jerarquía de la MIB es representado por un OID (Object IDentifier) único utilizado por SNMP para identificar y acceder a los elementos utilizando los tipos ASN1 comentados anteriormente para describirlos. Estos objetos descritos pueden ser objetos simples, escalares que representan una única instancia o tabulares que relacionan y almacenan varios escalares en tablas bidimensionales dentro de la MIB.

Estas MIB están definidas por organizaciones de estandarización pero suelen ser modificadas posteriormente por los diferentes fabricantes de acuerdo a las especificaciones de sus dispositivos.

Cada elemento dentro de la MIB es la hoja de un árbol que parte del nodo raíz (Root), a medida que descendemos por los subnodos del árbol vamos componiendo el OID del elemento.

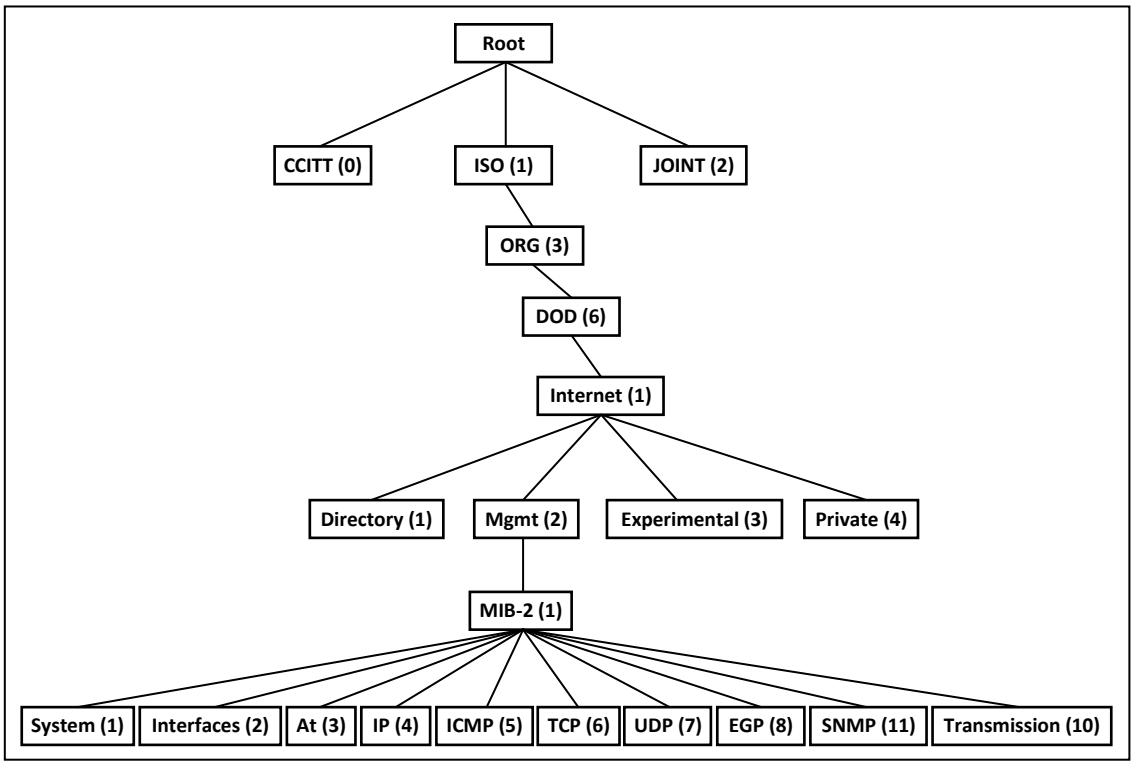


Figura 2.1. Representación en árbol de las entradas de una MIB y su OID.

En la figura 2.1 [10] observamos que el OID para un subelemento de SNMP sería:

Iso(1).org(3).dod(6).internet(1).mgmt(2).mib-2(1).snmp(11).Id_del_objeto(X)

O en forma de OID más abreviada: 1.3.6.1.2.1.11.X quedando de esta forma definido por su OID de forma unívoca no sólo cada elemento dentro de la MIB si no cada una de las categorías y subcategorías a las que pertenece dentro de su estructura, sea esta definida por un estándar o privada específica para la propia MIB.

2.1.2. SNMP Versión 1

Versión inicial del protocolo SNMP es una mejora del protocolo SGMP expuesta en el RFC 1175 [1]. Opera sobre IP y UDP a nivel de transporte para evitar la sobrecarga de red que podría provocar un servicio orientado a conexión como es TCP.

Al ser UDP un protocolo no orientado a conexión no hay seguimiento alguno de los paquetes enviados y por ello requiere un sistema de timeouts en las solicitudes para detectar paquetes perdidos debiendo ser el propio protocolo SNMP el que se encargue de resolver estas pérdidas.

Esto, sin embargo, puede ser un problema con el envió de traps desde los agentes de los dispositivos a los sistemas de monitorización ya que al no esperarse respuesta alguna del sistema el agente no será consciente de que el trap enviado no ha llegado a su destino y el sistema de monitorización no será consciente del evento ocurrido. Sin embargo, el reducido tamaño y consumo de red del protocolo UDP, compensa estás desventajas pues un protocolo de monitorización no debe interferir ni obstaculizar la comunicación en la red que monitoriza, sobre todo si esta red tiene una alta saturación pues un protocolo orientado a conexión como TCP insistiría en enviar los paquetes hasta que su destino le comunicase su correcta recepción saturando aún más la red y agravando cualquier problema que ya pudiera tener.

Define el concepto de "community" para identificar los tres diferentes grupos con acceso al agente SNMP y sus privilegios: "public" con derechos de solo lectura, "prívate" con derechos de lectura y escritura y "traps" que engloba a los destinatarios de los traps del agente. A pesar de esta definición como grupos los "community" son esencialmente claves, en función de la que se utilice al hacer una petición al agente esta se realizará con unos permisos u otros. Estas claves, con valores por defectos "public" y "prívate", se recomiendan cambiar en cada despliegue de sistemas, pero dado que viajan en texto plano dentro de los paquetes SNMP cualquier mensaje capturado por un intruso le concedería acceso al agente SNMP de los nodos. Esto es especialmente grave en caso del "community" de lectura-escritura, pues le concedería completo control sobre el mismo. Por ello se deben implementar sistemas externos de protección como firewalls, conexiones VPN, filtros de puertos e IPs, rotaciones de claves, etc. para dotar de seguridad a un despliegue con SNMP.

2.1.3. SNMP Versión 2

Con el paso del tiempo comprobó que la monitorización y gestión de redes era una tarea crítica y las deficiencias de SNMP se fueron haciendo cada vez más patentes, principalmente en materia de seguridad, no existía ningún método para autenticar una solicitud SNMP y únicamente los "community" que viajaban en texto plano servían para crear una falsa sensación de seguridad, por esta razón se publicaron una serie de RFCs bajo el nombre de "Secured SNMP". Estás definiciones no solucionaron los demás problemas de eficiencia y carencia de ciertas funcionalidades detectados, por lo que se procedió al desarrollo de SMP (Simple Management Protocol) el cual, sin embargo, no fue publicado como RFC. Tras la publicación de ambos documentos se crearon dos grupos de trabajo coordinados para la elaboración de una nueva versión de SNMP, cada uno de ellos encargados de diseñar el protocolo para cada una de las áreas cubiertas por las publicaciones anteriores, seguridad y funcionalidad.

Con esta segunda versión de SNMP se pretendían resolver los problemas de seguridad de SNMPv1 al tiempo que se le dotaba de mayores funcionalidades y se enriquecía el SMI de las MIBs.

Publicada como estándar en marzo de 1993 en la realidad no llegó a aceptarse como un estándar completo y no era 100% retrocompatible con la versión anterior pues el formato de mensajes y cabeceras había sido cambiado.

Tras unos años de uso el IETF encargó una revisión de esta versión del protocolo, resultado de esta revisión las modificaciones en materia de seguridad fueron desechadas por problemas de plazos y se utilizó una envoltura con el formato de mensajes SNMPv1 recuperando el uso del concepto de "community" creándose así la versión basada en "community" de SNMPv2 o simplemente SNMPv2C. Si se conservaron, sin embargo, tanto las mejoras funcionales y el SMI de las MIBs como la adición de más tipos ASN.1, macros y mejoras en las definiciones y manejo de tablas.

Algunas de las mejoras funcionales sobre la primera versión incluyen la adición de servicios para la consulta de grandes bloques de datos en una sola petición con el mandato GetBulk, o el mandato Inform que permite retransmitir información sobre los traps recibidos entre aplicaciones gestoras y que estás se comuniquen entre ellas.

Habitualmente las implementaciones, tanto de los agentes SNMP de los dispositivos como las aplicaciones de gestión, se realizan de forma que puedan atender y realizar peticiones de ambas versiones del protocolo ya sea directamente o mediante intermediaros capaces de traducir las solicitudes entre versiones. Aun así y pese a la revisión realizada por el IETF, la conformación de SNMPv2C y su implementación en los diferentes agentes de los dispositivos de red, la falta de un mecanismo de seguridad real dejaba la mejora del protocolo incompleta y con uno de sus principales objetivos, cubrir uno de los principales fallos del protocolo, inalcanzado, por todo ello el protocolo estándar de internet para monitorización y gestión de redes continuó siendo SNMPv1.

2.1.4. SNMP Versión 3

Es la última versión publicada de SNMP y resuelve las deficiencias dejadas por SNMPv2 en materia de seguridad. SNMPv3 no redefine completamente a SNMP si no que incorpora una serie de mecanismos de seguridad y un entorno para su utilización en conjunto con SNMPv1 y SNMPv2 así como aceptar revisiones futuras y mejoras en sus componentes. Estas mejoras añaden tanto seguridad como funcionalidades extra al protocolo, pero como contrapartida añaden complejidad a su uso e implementación.

La nueva arquitectura consta de un modelo de seguridad basado en usuarios USM (User-based Security Model) y un modelo de control de acceso basado en vistas VACM (View-based Access Control Model)

El modelo de control de acceso basado en vistas VACM proporciona dos importantes características: determina si un objeto a gestionar puede ser accedido para ciertas operaciones por un sistema remoto y hace uso de MIBs para fijar la política de acceso del agente SNMP permitiendo la configuración remota del dispositivo de una manera segura.

El modelo de seguridad basado en usuarios USM proporciona privacidad (cifrado) y autenticación a los mensajes del protocolo.

Para ello específica el uso de cuatro algoritmos criptográficos: uno para el cifrado de los mensajes, dos opciones para la generación de claves Hash y finalmente un algoritmo de autenticación de mensajes:

DES: Data Encryption Standard [11] para el cifrado de los mensajes, es un sistema de cifrado de clave simétrica o clave única.

MD5: Message-Digest Algorithm [12] como primera alternativa de función Hash o resumen para la autentificación de usuarios del USM.

SHA-1: Secure Hash Algorithm [13] Como segunda alternativa de función Hash para la autentificación del USM.

HMAC: Hash Message Authentication Code [12] que, en conjunto con las dos alternativas de función Hash, ofrece las dos opciones de autenticación disponibles en el protocolo: HMAC-MD5-96 y HMAC-SHA-96.

Estas características sumadas a las añadidas al protocolo original por la primera revisión del mismo, SNMPv2, conforman esta última versión SNMPv3 y así queda reflejado en el texto introductorio del borrador original publicado en el IETF [14] en 30 de Junio de 1988:

> "The SNMPv3 Working Group produced a design based on a modular architecture with evolutionary capabilities with emphasis on layering. As a result, SNMPv3 is SNMPv2 plus security and administration."

Tras todas estas revisiones podemos observar el árbol evolutivo del protocolo a lo largo de su vida:

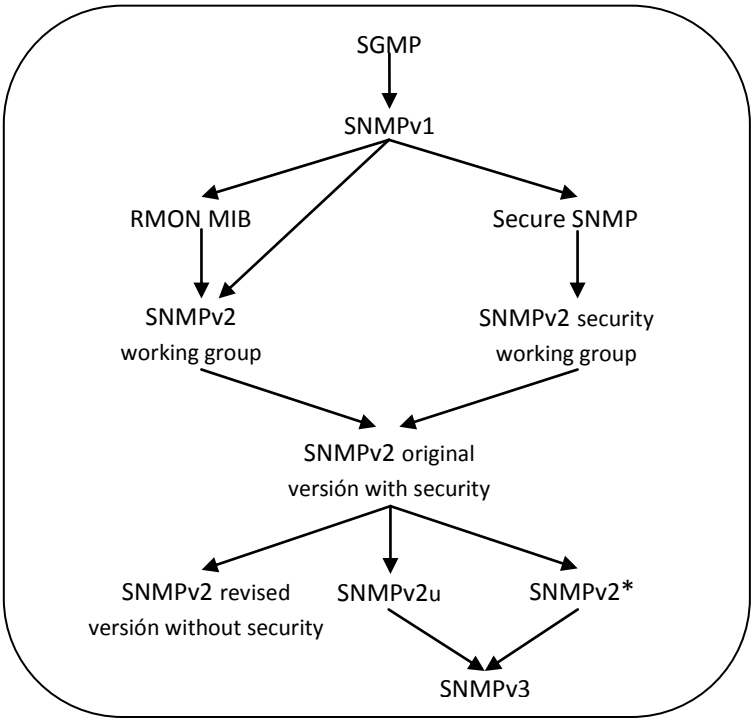


Figura 2.2. Evolución del protocolo SNMP.

2.2. Sistemas de monitorización

Existe un amplio catálogo de aplicaciones de monitorización de redes en el mercado tanto de código abierto como propietarias con diferentes ventajas. Las aplicaciones propietarias tienden a ser más completas y cuentan con el respaldo y garantía de las compañías desarrolladoras así como servicios técnicos especializados, existen tanto enfocadas al uso de redes en general, como Castle Rock SNMPc y, en otros casos, más enfocadas directamente a un fabricante o modelo de dispositivos como es el caso de Cisco Prime Network.

Estas aplicaciones están limitadas por sus propias licencias comerciales y dependen de sus desarrolladores para cualquier modificación o mejora y además suponen un desembolso económico considerable. Las aplicaciones de código abierto suelen ser de propósito general y cuentan con las ventajas de licencias de software libre como pueden ser libertad para copiar, modificar y distribuir, disponibilidad de código fuente e incluso gratuidad siempre que se respeten las condiciones de licencia bajo la que se presente. Suelen disponer de una gran comunidad de desarrolladores que implementan correcciones, mejoras y plugins que quedan disponibles para todos los usuarios del sistema.

A continuación se presentan algunas de estas aplicaciones, tanto de código libre como propietarias, genéricas y específicas, que se han analizado y tomado como referencia para estudiar los requisitos y funcionalidades de la aplicación GEMA:

2.2.1. Nagios

Nagios Core [15] es uno de los sistemas más populares de monitorización de redes de código abierto, licenciado bajo GPL está disponible de forma gratuita en su página web junto con todo el código fuente y un amplio catálogo de plugins, frontales y addons. Funciona bajo Linux y otras variantes Unix y soporta múltiples protocolos (SMTP, POP3, HTTP, NNTP, ICMP, SNMP, FTP, SSH, etc.), monitorización hardware del propio host del servidor, administración remota y notificación de alertas por correo electrónico y mensajería móvil.

Las diferentes APIs de que dispone permiten la integración con aplicaciones de terceros y la adición de plugins a la aplicación principal permitiendo extender así su funcionalidad según las necesidades de cada organización. Dispone también de capacidad de scripting en bash que puede ejecutarse en el servidor host de Nagios o en los sistemas monitorizados previa instalación en ellos de las librerías necesarias. Estas opciones junto con los catálogos de addons, plugins y la disponibilidad del código fuente permiten crear y configurar una instalación de Nagios de forma personalizada haciéndolo una aplicación muy flexible y adaptable.

Ofrece una vista centralizada de toda la red gestionada disponiendo toda la información de monitorización accesible a través de un interfaz web. Consta de mecanismos de gestión de usuarios y permisos tanto de visibilidad como de gestión para los mismos a diferentes partes de la red pudiendo controlar que usuarios acceden a que secciones o servicios de la infraestructura permitiendo utilizar la misma instalación de Nagios para monitorizar redes de diferentes clientes agrupándolas por grupos de usuarios. Permite también monitorización y gestión mediante túneles SSL cifrados y ofrece una consola SSH para su acceso remoto.

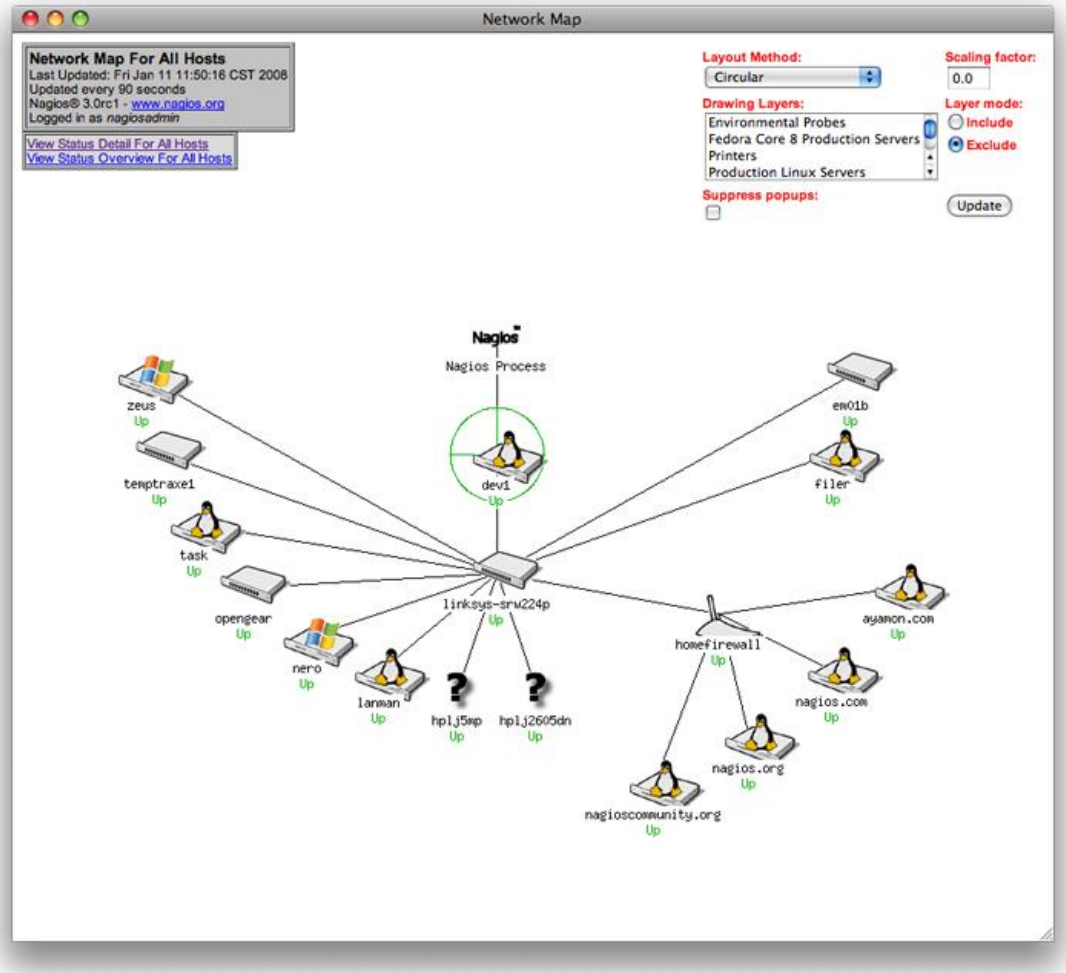


Figura 2.3. Presentación del estado de la red en Nagios.

Existe también una versión comercial, Nagios XI, que añade múltiples funcionalidades y servicios a la versión de libre distribución como un interfaz web más potente y configurable, definición de vistas para los usuarios, base de datos integrada, etc. Junto con esta versión se incluye soporte técnico profesional por una cuota anual. También se ofrece Nagios Fusión, una herramienta de visualización de información de monitorización avanzada integrable tanto con servidores Nagios Core como Nagios XI que aprovecha el acceso SSL de estos sin necesidad de configurar sistemas de comunicación adicionales.

2.2.2. Zenoss

Zennos Core [16] es una aplicación de gestión y monitorización de redes y servidores desarrollada en Python basada en el servidor de aplicaciones web Zope (Uno de los principales proyectos impulsores del lenguaje de programación Python). Distribuida bajo licencia GPL para diferentes plataformas Unix y Linux, tiene soporte multiprotocolo para monitorización de dispositivos, servicios (HTTP, POP3, NNTP, SNMP, FTP) y máquinas virtuales, mecanismos de descubrimiento, recursos del servidor local, gestión de eventos, etc. Es, además, compatible con el formato de plugins de Nagios permitiendo así aumentar sus funcionalidades con todo el software libre desarrollado para esta plataforma y disponible en la red.

Una de sus características más destacables es la capacidad de autodetectar los dispositivos conectados a la red mediante mecanismos de descubrimiento de forma dinámica a medida que estos se van conectando a la red para proceder automáticamente a su monitorización.

Ofrece un frontal web con información general y detallada de cada dispositivo permitiendo organizarlos en grupos, asociarlos a ubicaciones geográficas y asociarles etiquetas de estado. Genera gráficas e historiales de eventos, informes de rendimiento, avisos mediante correo electrónico y mensajería móvil, consta de soporte multiusuario con gestión de permisos para limitar el acceso de cada uno a los dispositivos y recursos asignados, dispone también de integración con sistemas de terceros como Google Maps y ofrece APIs de servicios web para su utilización con otros sistemas.

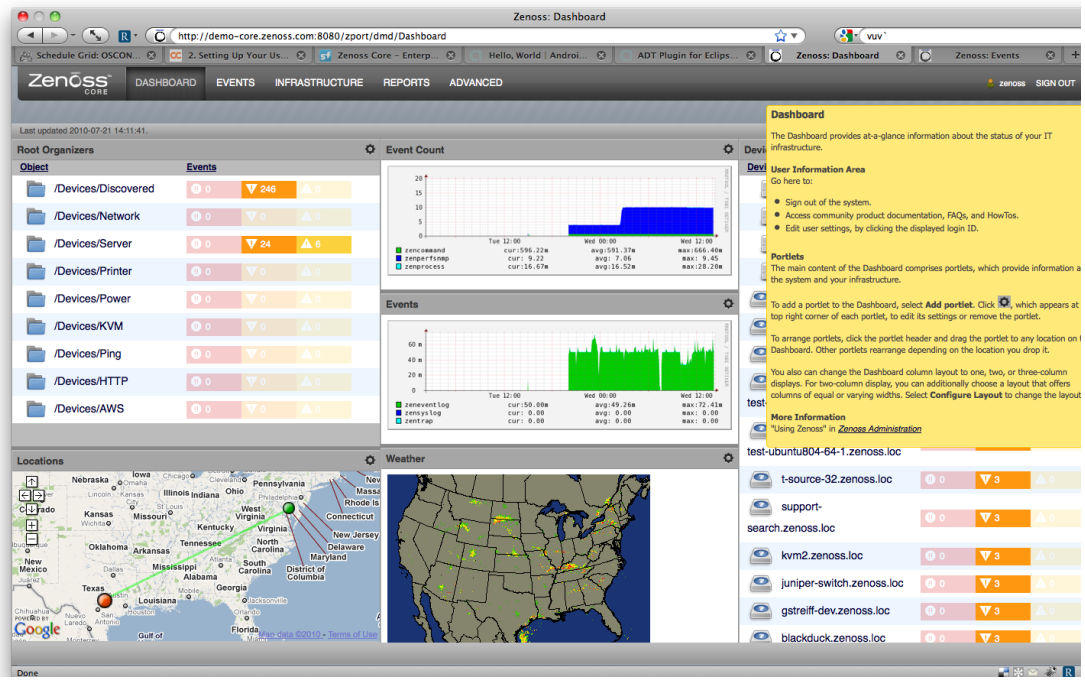


Figura 2.4. Página de resumen de Zenoss.

Dispone de una arquitectura de scripting, denominada ZenPacks, que incluye tres categorías de plugins: Core, desarrollados por Zenoss Inc; Community, desarrollados por la comunidad de usuarios y distribuidos gratuitamente mediante el formato .egg de Python y Commercial, desarrollados por Zenoss y distribuidos comercialmente. Esto le dota de una amplia gama de plugins disponibles de forma nativa. Además soporta el formato de plugins de Nagios lo cual aumenta enormemente su catálogo de funcionalidades ampliables. También admite scripting en Python lo que permite añadir funcionalidades a medida al sistema en el propio lenguaje nativo de la aplicación. Todas estas características hacen de Zenoss un sistema altamente configurable y personalizable a la medida de la empresa.

Tiene soporte directo para bases de datos MySQL (Motor de bases de datos relacional de código abierto) y RRDTool (Herramienta para el manejo de Bases de datos rotatorias) [17] para la generación de informes, estadísticas e inferencias y, al igual que Nagios, dispone de una versión comercial, Zennos Enterprise, que incluye funcionalidades adicionales y soporte técnico.

2.2.3. Castle Rock SNMPc

SNMPc [18] es una aplicación de monitorización de redes y servidores propietaria de Castle Rock Computing, pionera en sistemas de monitorización bajo entornos Windows, independiente de fabricantes con soporte SNMP hasta su versión 3, IPv6, despliegue distribuido y acceso a la información vía web.

La aplicación utiliza un modelo de agentes monitores distribuidos, cada agente monitor se encarga de recabar la información de un subconjunto de la red o redes a monitorizar y comunica únicamente los cambios a monitor de nivel superior. Pudiendo crearse varios niveles en la jerarquía hasta el sistema principal encargado de proporcionar la información hacia los usuarios. De esta forma no solo se crea una estructura fácilmente ampliable y jerarquizada si no que se hace un uso más eficiente del ancho de banda de las redes no saturándolas de comunicaciones de monitorización innecesarias y permite escalar adecuadamente los recursos hardware del sistema de monitorización.

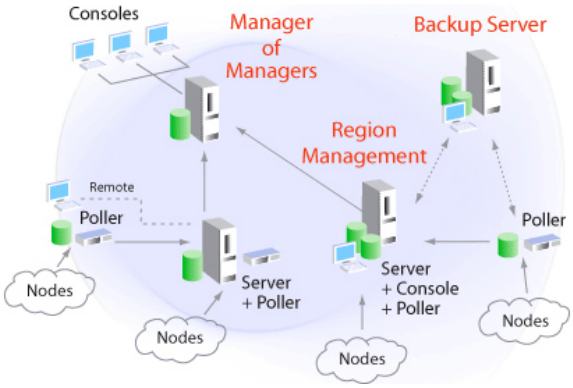


Figura 2.5. Modelo distribuido de SNMPc.

Dispone de un sistema de notificaciones por correo y notificaciones emergentes, visuales y sonoras en los terminales de consulta, además permite definir filtros de eventos basados directamente en las entradas de los ficheros de log. El sistema también es capaz de generar alertas de forma automática, a medida que monitoriza la red va aprendiendo el comportamiento habitual de las variables y definiendo un patrón, generando alertas cuando los valores de algunos parámetros se desvían excesivamente del patrón aprendido.

La monitorización en remoto se realiza a través de la aplicación cliente JAVA, con soporte multiplataforma, para las funciones básicas o mediante el cliente Windows con completa funcionalidad. El número máximo permitido de consolas cliente conectadas simultáneamente dependerá de la licencia adquirida. Incorpora, al igual que las aplicaciones estudiadas anteriormente, soporte multiusuario con gestión de privilegios y permisos sobre diferentes partes de las redes monitorizadas.

El scripting de visualización se realiza mediante la herramienta propietaria BitView y dispone de varios interfaces de programación para la implementación de scripts y da soporte de loggin, exportación de información geográfica y estadísticas para los sistemas de bases de datos estándar del mercado

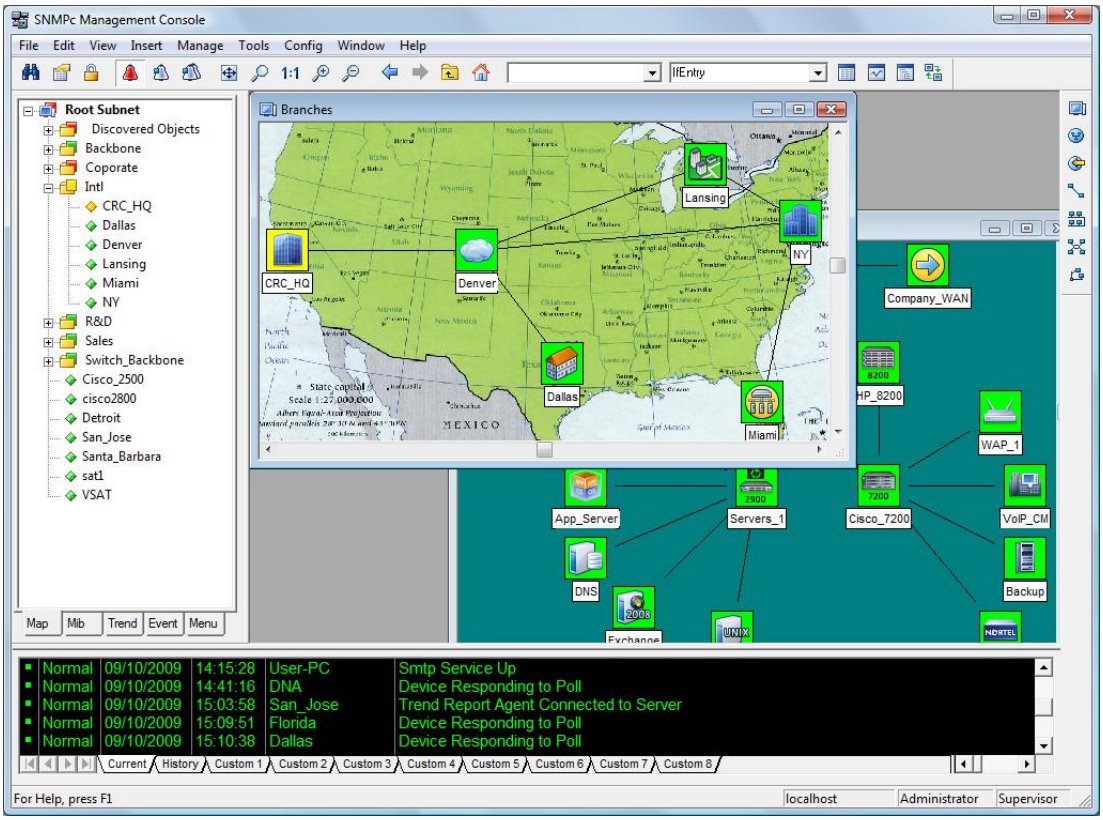


Figura 2.6. Consola de configuración de SNMPc.

Existe una versión orientada a la mediana y pequeña empresa, SNMPc Workgroup Edition de menor coste y una versión completa, SNMPc Online, que incluye, además de la versión Enterprise, diversas herramientas de visualización de alto nivel, programación de notificaciones de correo y representación gráfica de las redes entre otras funcionalidades añadidas.

2.2.4. Cisco Prime

Cisco Prime [19] es un conjunto de aplicaciones integrado de monitorización y gestión de red, utilizables de forma independiente o en conjunto, que cubre las diversas funcionalidades de un sistema completo de gestión y monitorización, incluidas análisis de calidad de servicio, de costes, etc. Esta "suite" incluye, aparte de una aplicación, Cisco Prime Central, que centraliza toda la información otras dos específicas para la gestión y monitorización:

Cisco Prime Network: Es una aplicación de gestión y monitorización de redes multiservicio con soporte tanto específico para los productos Cisco como para sistemas de terceros. Dispone de visualización gráfica de la red a diferentes niveles de detalle, sistema de configuración gráfico y diseño gráfico y posibilidad de scripting con más de 200 scripts pre programados, representación en tiempo real de alertas, eventos y cambios en la configuración. Posee un sistema de resolución de fallos basado en la topología y detección automática de causas reduciendo el número de alertas en la red al detectar duplicidad en las mismas. Está preparado y optimizado para redes en las que se involucran diferentes tecnológicas y productos de diferentes fabricantes.

Cuenta con un SDK (Software Development Kit) y soporte para comunicación con aplicaciones de otros fabricantes como InfoVista Vitalnsight e IBM Tivoli Netcool.

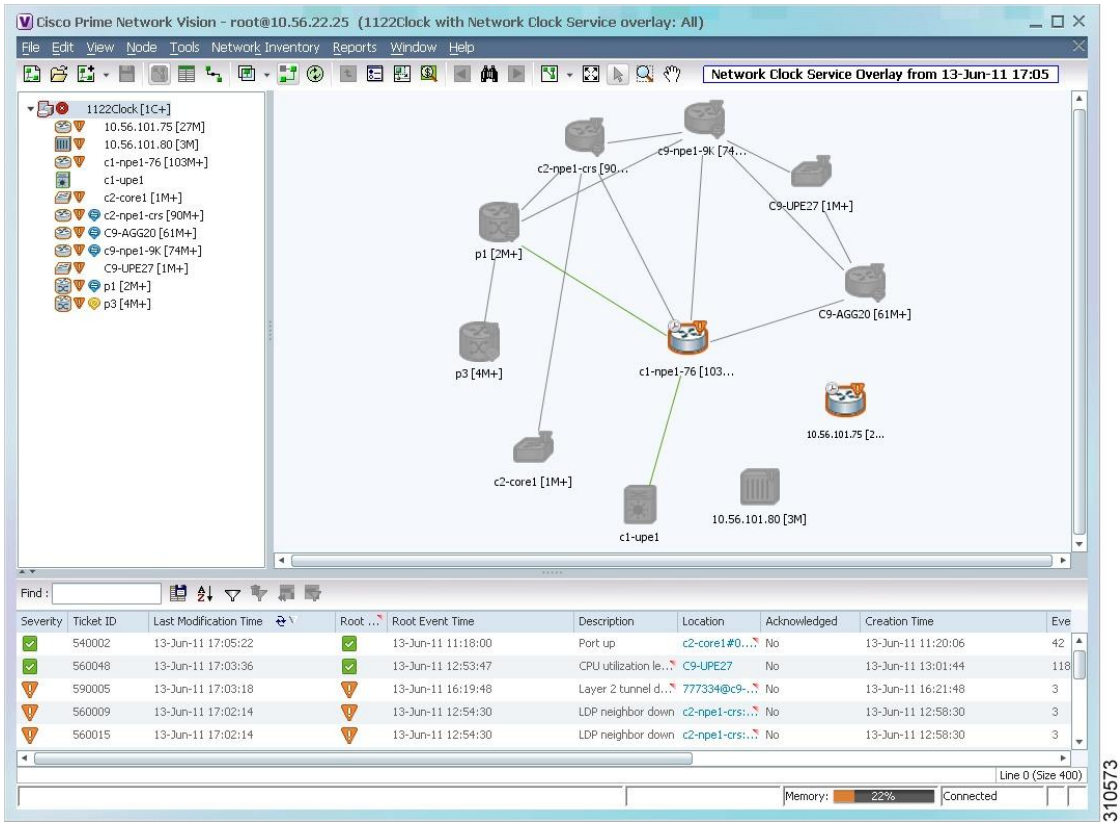


Figura 2.7. Ventana de visualización de Cisco Prime Network.

Cisco Prime Optical: Es un sistema de gestión de redes ópticas con soporte para tecnología TDM (Time Division Multiplexing), WSON (Wavelength Switched Optical Networks), DWDM (Dense Wavelength-Division Multiplexing) y SONET / SDH (Synchronous Optical Networks Synchronous Digital Hierarchy) capaz de controlar hasta 5000 elementos de red óptica simultáneamente. Dispone de soporte tanto para Linux como para Cisco UCS (Cisco Unified Computing System) y sigue los estándares CORBA TMF 814v3 y NBI (Northboun Interface) para la integración con otros sistemas.

Cisco Prime se distribuye con varias licencias para los diferentes elementos y dispositivos, la licencia básica es necesaria para cada nodo de monitorización y están disponibles otras licencias mayores que incluyen configuración, seguimiento y seguro. Una licencia para los sistemas de notificación y configuración automática en los cuales los nuevos nodos pueden solicitar su configuración y software y, finalmente, una licencia de conformidad que proporciona acceso a informes y estándares regulatorios.

2.2.5. Zabbix

Zabbix [20] es una aplicación de monitorización de código libre completamente gratuita bajo licencia GPL2. No dispone de una versión de pago ya que todas sus características están incluidas en la versión libre.

Esta aplicación tiene soporte multiplataforma y no se ejecuta sobre una máquina virtual o entorno como Java, .Net o Python.

Permite una implantación distribuida tanto mediante proxies que extienden un servidor central como mediante diferentes servidores Zabbix comunicándose entre ellos.

Dispone de mecanismo de descubrimiento de dispositivos de red automáticos a bajo nivel. Soporta monitorización por SNMP, controles ICMP, IPMI y de servicios web como FTP, SSH, HTTP, DNS, etc. sin necesidad de instalar software específico en los servidores.

Es capaz de mantener hasta 100,000 dispositivos monitorizados. Dispone de envío de notificaciones por E-mail, SMS, y sistemas de mensajería instantánea.

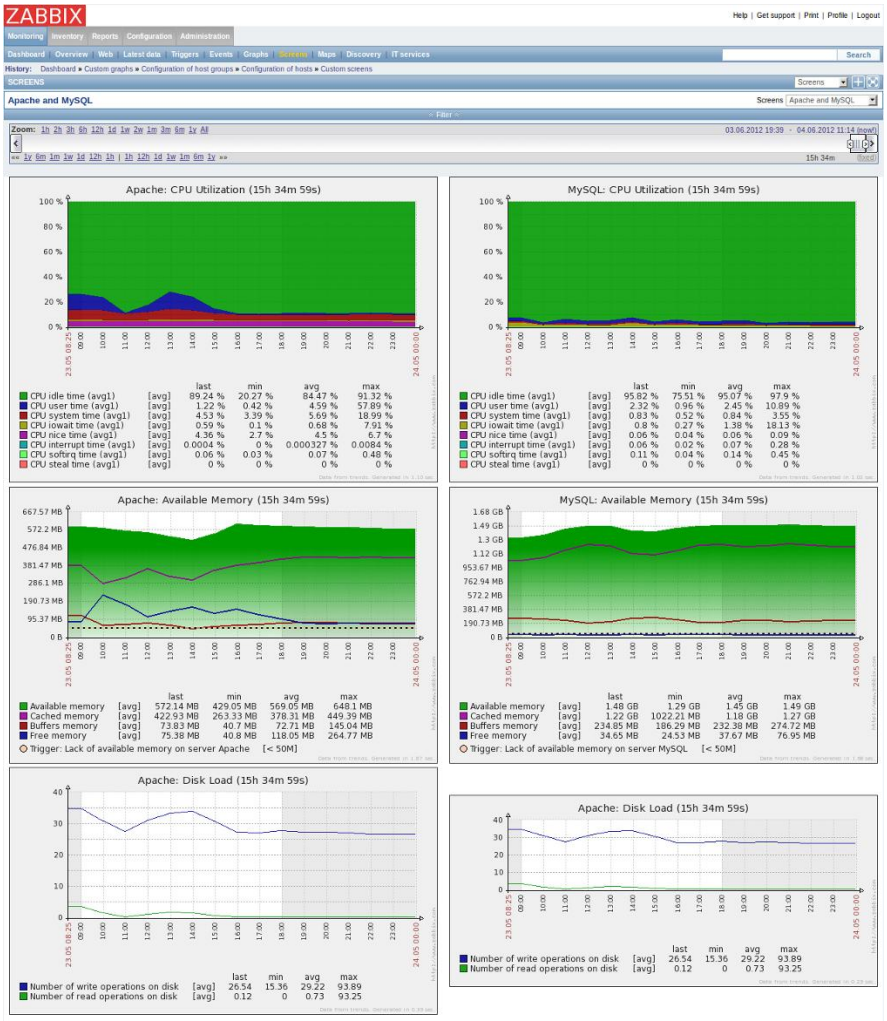


Figura 2.8. Ventana de Monitorización de Zabbix.

Dispone de un software agente a instalar en máquinas servidoras a monitorizar para permitir el acceso a información local como recursos hardware tales como carga de la CPU, utilización de la red y espacio en disco.

El almacenamiento de la información de monitorización se realiza en base de datos. Zabbix es compatible con bases de datos MySQL, PostgreSQL, Oracle o SQLite.

El interfaz de usuario es vía web con controles de seguridad basados en SSL, resistencia a ataques por fuerza bruta, listas de direcciones IP con permisos para los diferentes componentes, usuarios y grupos de permisos, etc. Tiene soporte para gestión de usuarios y permisos utilizando diversas tecnologías como servidores LDAP o Radius.

Dispone de capacidad de integración con scripts en diferentes lenguajes como Ruby, Python, Perl, PHP, Java o Shell-script que permiten extender sus funcionalidades sin requerir un entorno o lenguaje específico de programación.

2.2.6. Cacti

Cacti [21] es un front-end para RRDTool [17] desarrollado completamente en PHP. Se distribuye de forma gratuita bajo la licencia GNU-GPL. Es básicamente una herramienta de procesamiento y visualización de información almacenada en bases de datos y ficheros round robin.

Dispone de una gran variedad de formatos de representación mediante gráficos y permite la creación de plantillas de gráficos para tipos de dispositivos específicos. Existen en la web una amplia gama de plantillas de gráficas y scripts para la configuración de la aplicación.

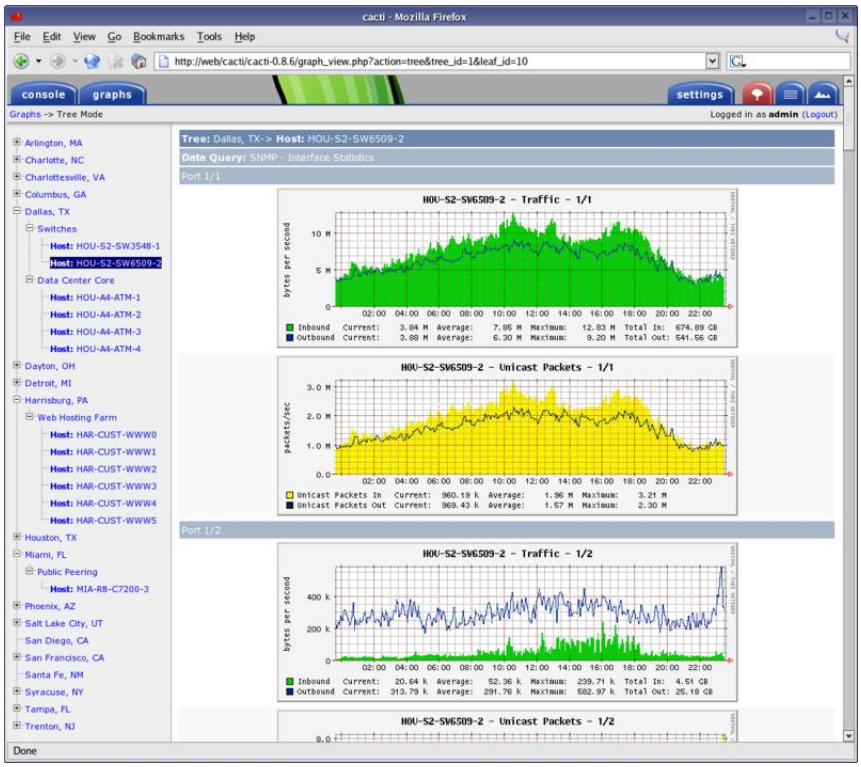


Figura 2.9. Ventana de gráficos de Cacti.

Cacti estructura su funcionamiento en tres pasos diferenciados: Obtención de los datos, almacenamiento y representación.

Para la recopilación de datos utiliza un sistema de captura ejecutado por el planificador del sistema, por ejemplo Crontab en sistemas Unix. Este sistema de captura soporta la utilización de scripts y ejecutables.

Dispone de soporte para SNMP que permite monitorizar dispositivos habilitados para este protocolo. En cada ejecución del mecanismo de captura estos scripts y solicitudes SNMP son realizadas para capturar los datos.

Los resultados de estos scripts y peticiones son capturados y almacenados en la base de datos y ficheros round robin a los que accederá Cacti para generar las gráficas.

Para la generación de las gráficas Cacti dispone de diversos modelos programables de gráficas en los que es posible procesar la información y decidir la forma de representarla. Permite la introducción de operaciones matemáticas en el lenguaje CDEF de RRDTool para la generación de estos gráficos.

También dispone de mecanismos de gestión de usuarios con diferentes configuraciones y permisos para cada uno.

2.2.7. Centreon

Centreon [22] es un front-end de Nagios gratuito y de código libre distribuido bajo la licencia GPL2 por la compañía Merethis. Puede operar directamente sobre Nagios o sobre Centreon Engine. Centreon Engine es una modificación de Nagios basada en su versión 3.2.3 realizada por Merethis en la que trata de mejorar la eficiencia de este y dotarle de nuevas tecnologías.

Posee un diseño modular que permite la adición y modificación de sus funcionalidades ya sea mediante plugins, reprogramación o incorporación de nuevos módulos.

Entre sus características se encuentra el soporte para monitorización por SNMP, captura automática de traps, planificación de apagados de mantenimiento, servicio de consultas activas y pasivas.

Permite una implantación distribuida con servidores de reemplazo para caso de fallo. Permite replicación de servicios web, bases de datos y motores de monitorización para mantener un servicio de alta disponibilidad.

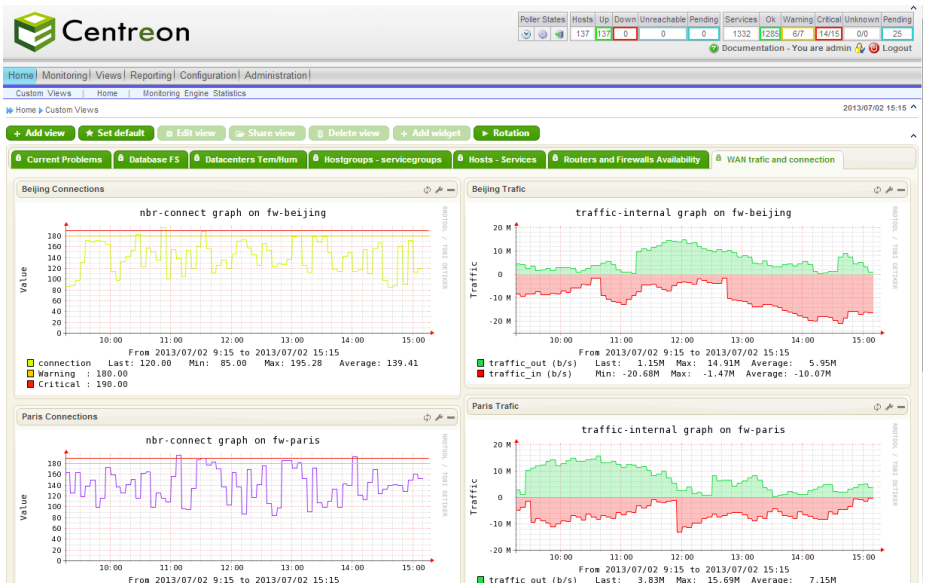


Figura 2.10. Ventana de monitorización de Centreon.

Tiene soporte para usuarios y grupos de usuarios con gestión de permisos y restricciones. Soporta Autentificación mediante LDAP y realiza logs de acciones de los usuarios.

Centreon también dispone de una serie de aplicaciones adicionales que incorporan funcionalidades extra como business intelligence o mapeado de redes.

Estas aplicaciones pueden descargarse e instalarse por separado o en conjunto en la suite Centreon Enterprise Server y se distribuyen igualmente de forma gratuita bajo licencia GPL2.

2.2.8. Tabla comparativa

Tabla comparativa que presenta comparativamente las principales características de los sistemas de monitorización y gestión presentados.

Nombre	Nagios	Zennos	Castle Rock	Cisco Prime	Zabbix	Cacti	Centreon
Gráficas	✓	✓	✓	✓	✓	✓	✓
Grupos lógicos	✓	✓	-	-	✘	✓	✓
Estadísticas	✓	✓	✓	✓	✓	✓	✓
Predicción de estadísticas	✓	✓	-	✓	✓	-	✓
Auto descubrimiento	✓	✓	✓	✓	✓	✓ Plugin	✓
Agentes	✓	✓	-	-	✓	✓	✓
SNMP	✓ Plugin	✓	✓	✓	✓	✓	✓
Syslog	✓	✓	✘	✓	✓	✓	✓
Scripts externos	✓	✓	✓	✓	✓	✓	✓
Complementos (plugins)	✓	✓	✘	✓	✓	✓	✓
Alertas	✓	✓	✓	✓	✓	✓	✓
Aplicación web	✓ Visualización	✓	✓ Visualización	✓ Visualización	✓	✓	✓
Monitorización distribuida	✓	✓	✓	✓	✓	-	✓
Almacenaje de datos	SQL	RRDTool y MySQL	-	-	SQL	RRDTool y MySQL	SQL
Licencia	GPL	GPL	Comercial	Comercial	GPL	GPL	GPL
Mapas	-	✓	✓	-	✓	✓ Plugin	✓ Plugin
Seguridad	-	✓	✓	✓	✓	-	✓
Eventos	-	✓	✓	✓	✓	-	✓

Figura 2.11. Tabla comparativa.

2.3. Scripting para sistemas de monitorización

Los sistemas de scripting son mecanismos mediante los cuales se pueden alterar o extender las funcionalidades de una aplicación mediante la programación. Por lo general consisten en una serie de órdenes a ejecutar al activarse el script tras aparecer un determinado evento o cumplirse una serie de condiciones, sin embargo es posible encontrarse sistemas de scripting capaces de implementar comportamientos más complejos incluyendo evaluación de condiciones y control de ejecución mediante bucles y otras estructuras de programación.

Los lenguajes de scripting para aplicaciones pueden ser, desde lenguajes específicos desarrollados y orientados expresamente para la programación de la aplicación mediante un conjunto reducido de ordenes sencillas, hasta lenguajes de propósito general embebidos en la propia aplicación con acceso a toda la información y funcionalidades de la misma, pasando por lenguajes diseñados específicamente para las labores de extensión de aplicaciones pero desarrollados de forma genérica para su uso con múltiples aplicaciones.

Los lenguajes específicos tienen la ventaja de estar desarrollados desde su concepción teniendo en mente la aplicación que se va a controlar con ellos, esto les permite ser muy sintéticos en su uso permitiendo el acceso y control de los recursos de la aplicación de una forma eficiente y directa. Igualmente permite limitar que partes de la aplicación pueden ser afectadas evitando así que se pueda alterar el comportamiento de recursos críticos o acceder a información o recursos no permitidos. La principal contra de estos lenguajes es que la reusabilidad del código queda reducida al ámbito de la aplicación y, que en caso de precisarse, implementar un comportamiento más allá del planteado inicialmente para el lenguaje puede resultar muy complicado cuando no imposible. En muchos casos, estos lenguajes específicos corresponden a un subconjunto de un lenguaje de propósito general conocido o basan su sintaxis en ellos, de forma que se atenúa la limitación de reusabilidad de código al tiempo que se facilita el acceso a ellos al seguir un modelo de lenguaje generalmente conocido.

Los lenguajes de scripting de uso general son leguajes diseñados específicamente para esta labor pero sin enfocarse o limitarse a una aplicación concreta, de forma que el código de estos lenguajes es fácilmente portable y reutilizable debido a su estandarización. Así se facilita el acceso a la programación de scripting en las aplicaciones al realizarse en un lenguaje estandarizado y conocido. Existen diversos lenguajes en esta categoría algunos de los más habituales son ECMAScript (Java Script, ActionScript,…), Tcl, LUA.

Los lenguajes de propósito general, aparte de contar con las ventajas de los lenguajes estándar de scripting, tienen la ventaja de ser lenguajes más completos no limitados al ámbito de la extensión de funcionalidades si no que están diseñados para el desarrollo completo de aplicaciones. Esto implica que, a priori, no existe un límite establecido por la aplicación al comportamiento que se puede implementar, si se limitará el acceso a la información y recursos críticos de ser necesario estableciendo que servicios son accesibles por el código del script. Mediante estos leguajes es virtualmente posible implementar cualquier software dentro de la aplicación que los incorpora. Algunos de los lenguajes de uso general más habituales en este ámbito son Perl, Python, Ruby.

En los sistemas de monitorización el scripting es utilizado para diversas tareas, principalmente en la atención de eventos, de forma que el sistema pueda analizar la información recibida de la red tanto forma síncrona mediante la monitorización como asíncrona por la recepción de eventos desde los dispositivos. Este análisis permitirá sintetizar las alertas en avisos de más alto nivel, como agrupar una serie de alertas en un solo aviso del problema que las ha causado, tomar actuaciones automáticas e incluso generar informes exhaustivos y enviarlos a destinatarios definidos. Otro ámbito de uso habitual es en la capa de presentación, permitiendo definir mediante un lenguaje dedicado a ello el aspecto y la información a mostrar en el interfaz con el usuario.

El uso más básico de scripting para monitorización es la creación de scripts de consola (Shell Scripting) que al ser invocados por el administrador o de forma recurrente mediante un servicio tipo CRON, ejecuta una serie de peticiones definidas, analiza ficheros de log o realiza labores de gestión establecidas. Mediante estas colecciones de scripts se puede programar comportamientos específicos dentro de un equipo de monitorización a nivel de sistema sin necesidad de ninguna aplicación específica de más que los recursos y servicios ofrecidos por el sistema operativo y aplicaciones de línea de comandos. La mayor parte de estos scripts trabajan a nivel de texto, leyendo los resultados de los mandatos solicitados y parseándolos mediante expresiones regulares para tomar unas actuaciones u otras.

Una vez ya dentro de una aplicación de monitorización se pueden encontrar diferentes niveles de complejidad de scripts, desde simples scripts con una serie de órdenes secuenciales a ejecutar cuando se cumple una condición dada hasta complejos desarrollos software que incorporen nuevas funcionalidades a la aplicación. Estos scripts pueden ir enfocados tanto a la automatización de las labores de gestión como al procesado de información presentando la información de red de una forma mucho más accesible de cara al administrador o usuario emitiendo informes más elaborados.

Habitualmente las aplicaciones de monitorización, como las analizadas anteriormente, ofrecen esta capacidad de extensión utilizando lenguajes tanto específicos como estandarizados para este uso incluso, como en el caso de Zenoss, con un lenguaje de propósito general como es Python. De esta forma permiten no solo la implementación de scripts sencillos que reaccionan ante los eventos o estados de la red si no que es posible desarrollar completos paquetes de funcionalidades añadidas a la aplicación principal que, posteriormente, pueden ser distribuidos en forma de plugins incluso a otras aplicaciones diferentes gracias a estos lenguajes estandarizados, como es el caso, nuevamente, de Zenoss que soporta el sistema de plugins de Nagios.

Es posible implementar comportamientos complejos que se ejecuten de forma automática no monitorizados por los administradores, es decir, que la aplicación controle directamente la red tomando decisiones basándose en la información de la red y una serie de premisas definidas en una base de conocimiento para realizar un diagnóstico sin intervención humana. Parece, sin embargo, que no es la tendencia más habitual y el scripting se enfoca más en la dirección de automatizar tareas de procesado de la información a mostrar o realizar solicitudes de información a los dispositivos como respuesta a ciertos eventos para generar avisos a los usuarios más elaborados.

2.4. Persistencia

La persistencia es el medio por el cual los datos en tiempo de ejecución de una aplicación son almacenados de forma estática para su posterior recuperación y uso por otra ejecución de la misma aplicación u otras diferentes. Existen diferentes técnicas de persistencia más o menos adecuadas según los diferentes tipos de datos, usos o situaciones. En los sistemas de monitorización existen principalmente dos grandes categorías o grupos de datos con las que trabajar en reflejo de las dos grandes responsabilidades de este tipo de software: administración y monitorización.

La primera, administración, es toda la información relativa a la red administrada: su geometría, componentes y características, conexiones, configuraciones, relaciones jerárquicas, disposición geográfica, protocolos, etc. El otro gran grupo es la información propia resultado de la monitorización, los valores leídos de cada variable en cada instante junto con todos los valores inferidos de ello.

Existen otros grupos de información más generales como son la referente a las acciones o eventos de la aplicación, los datos de gestión y control de usuarios, configuraciones propias del sistema e incluso, para los sistemas multiprotocolo y en las aplicaciones que admiten respuesta automática a estados o eventos en la red monitorizada, información relativa a los diferentes protocolos y a la base de conocimiento utilizada para tomar decisiones de actuación en la red.

2.4.1. Grupos de datos

En función del conjunto de datos con el que estemos trabajando deberemos tener en cuenta sus características para elegir convenientemente el mecanismo de persistencia más adecuado.

En los grupos anteriormente citados podemos observar las siguientes:

- **Datos de monitorización**: Dado su carácter de "tiempo real" son un conjunto de información que ha de contar con un historial, de forma que se pueda obtener y analizar el estado de la red en un instante o intervalo de tiempo a posteriori, es por ello, además, un conjunto de datos que aumenta rápidamente de tamaño. También es un conjunto de datos para el que posiblemente se requiera acceso por terceras aplicaciones como puedan ser sistemas de análisis estadístico o de representación gráfica de cara a los administradores o usuarios.
- **Descripción de la Red**: La información relativa a la red es fácilmente estructurable, abstraible y jerarquizable, además suele ser bastante estática, con pocas variaciones en el tiempo una vez introducida, principalmente altas de nuevos elementos. También ha de ser, en la medida de lo posible, independiente de la aplicación, principalmente de cara a los cambios internos del modelo o clases que lo implementen debidos a correcciones o mejoras del diseño o la aplicación.
- **Los datos de configuración**: Información y variables locales del sistema e instalación, protocolos, ficheros de configuración, bases de conocimientos, etc. Es información principalmente estática introducida por los administradores en el momento de configuración de la red en "frio".

Definen los patrones de comportamiento de la aplicación. En este grupo también se engloban las MIBs de SNMP con las definiciones de los dispositivos a gestionar.

- **Logs de acciones y eventos de la aplicación**: Son colecciones secuenciales de información ordenada temporalmente. Esta información va dirigida principalmente a los administradores por lo que debe ser almacenada en un formato legible y accesible externamente para poder analizar el comportamiento del sistema.
- **Información de Control**: Datos relativos a usuarios, conexiones, control de acceso, etc. Esta información suele estar estructurada a modo de tablas, almacenando datos y permisos específicos de cada entrada, usuario o sistema, que interactúa con la aplicación. Consta de una parte bastante estática relativa a los datos descriptivos de cada entrada: nombres, permisos, etc. y una parte dinámica que representa las acciones tomadas por cada elemento: fechas de conexión, acciones y/o modificaciones realizadas, etc.

2.4.2. Sistemas de persistencia

En función de las características de los datos a almacenar deberemos elegir el sistema de persistencia más adecuado. Para ello analizamos diferentes sistemas y sus características contrastándolos con los grupos de datos que maneja la aplicación:

- **Ficheros de texto secuenciales**: Son ficheros en texto plano escritos por líneas y de recorrido secuencial. Su actualización se realiza añadiendo líneas al final del fichero pudiendo crecer indefinidamente de forma ordenada en el tiempo. Son fácilmente accesibles tanto por aplicaciones externas como directamente por los administradores. Por estas características son el sistema más ampliamente utilizado para el almacenamiento de los logs de eventos y acciones del sistema y aplicaciones.

  Para poder controlar el volumen ocupado en disco por estos ficheros, que crecen indefinidamente a medida que el sistema genera eventos y realiza acciones, se utilizan técnicas de rotación. Para ello se define un tamaño máximo de fichero y un número de ficheros máximo, cuando el primer fichero alcanza su capacidad máxima es renombrado con un código definido y se continúan registrando entradas en un fichero nuevo. Cuando el sistema de logs alcanza el límite de ficheros establecido se descarta el más antiguo, o se desplaza a otra localización de almacenamiento masivo para su conservación, quedando de esta forma un hueco libre, se renombran los demás ficheros de forma adecuada y se continúa en uno nuevo, controlando así el tamaño máximo que los ficheros pueden ocupar en el sistema de monitorización.
- **Ficheros estructurados**: A diferencia de los ficheros secuenciales, los ficheros estructurados constan de una estructura prefijada para almacenar los datos. Existen varias definiciones o lenguajes estándar como XML [23] o la estructura definida en SMI (Structure of Management Information) para las MIBs en SNMP. Esta estructura permite definir una serie de algoritmos o reglas de lectura para obtener la información almacenada y tratarla de la manera que le corresponda.

Ya sea utilizando un formato estándar definido o diseñando uno específico para la aplicación este tipo de ficheros son los más adecuados para almacenar las configuraciones tanto de la aplicación como de los diferentes módulos. También resultan adecuados para definir reglas de comportamiento en forma de scripts de código al ser los lenguajes de programación lenguajes estructurados según una serie de reglas definidas por sus gramáticas.

- **Ficheros serializados**: Los ficheros serializados son el resultado de aplicar un proceso de aplanado o "marshalling" sobre las estructuras en memoria de una aplicación, de forma que posteriormente puedan ser leídos para reconstruir fielmente la misma estructura en memoria en otra ejecución o en otra aplicación. De esta forma se podrán crear diferentes estructuras y almacenarlas para su posterior recarga ahorrando el proceso de diseño de las mismas en cada uso.

  Es una técnica ampliamente utilizada para dotar de persistencia a modelos de objetos, tanto para su almacenamiento como para su transmisión en red a otras aplicaciones. Muchos lenguajes de programación, como: Java, Delphi, C#, Perl, Python por citar algunos, incluyen librerías para realizar este proceso de forma nativa, lo que la hace la técnica más adecuada para almacenar la estructura de red en memoria durante la ejecución del sistema de monitorización.

- **Ficheros binarios**: Un fichero binario simplemente es aquel en que la información ha sido codificada antes de almacenarse en lugar de escribirse directamente en texto plano. Cualquiera de los modelos de ficheros anteriormente comentados pueden ser escritos en texto pero su almacenamiento binario permite compactar los datos aprovechando el espacio, de forma que el fichero final no solo no es legible directamente por usuarios si no que requiere de algún mecanismo específico para recuperar los datos. Por esto resulta una técnica recomendable a la hora de generar ficheros, tanto de cara a ahorrar espacio en el sistema como para asegurar cierto nivel de seguridad en los mismos. Pudiéndose, además, aplicar técnicas de cifrado, firmado etc. sobre los mismos.

- **Bases de datos relacionales**: Las bases de datos relacionales son el modelo de persistencia más utilizado actualmente para cualquier tipo de datos estructurado. En ellas se definen una serie de entidades con atributos y unas relaciones que posteriormente son transformadas en tablas que se almacenan en el sistema gestor de bases de datos. Estos sistemas definen operaciones estándar de creación, lectura, escritura y borrado de datos, CRUD (Create, Read, Update, Delete) además de otras funcionalidades de gestión del sistema gestor.

  La información almacenada en ellas es accesible de forma estándar por cualquier sistema que las conecte. Los sistemas gestores están desarrollados de forma independiente y altamente optimizados para la gestión, acceso y almacenamiento de información, estando además dotados de sus propios sistemas de comunicación en red.

  Resultan por ello un método muy adecuado para almacenar información estructurada según el modelo relacional, entidades con atributos y relaciones similar al modelo de clases de la programación orienta a objetos o estructurada en tablas.

Así mismo también resulta adecuado para cualquier otro tipo de información no estructurada como puedan ser algunos ficheros al permitirse almacenar estos como el valor de un campo en una de las tablas, resultando totalmente independiente de la aplicación.

- **Bases de datos Round Robin**: Las bases de datos round robin están específicamente diseñadas para almacenar historiales de datos variables en el tiempo obtenidos a intervalos regulares. Son capaces de interpolar datos tanto internos a los intervalos como datos no recibidos y definen un volumen máximo de almacenamiento de una forma similar a los ficheros rotatorios expuestos anteriormente. Combinando estas características resultan un mecanismo de persistencia muy adecuado a información de monitorización de todo tipo y por ello es ampliamente utilizado por estos sistemas siendo una de las características listadas en las comparativas de sistemas de monitorización.

  RRDTool [17] es el estándar de código abierto de implementación de este sistema de almacenamiento, es multiplataforma y dispone de librerías para su uso bajo diferentes lenguajes de programación.

2.5. Paradigma de programación a emplear: Orientación a Objetos

Dentro de los diferentes paradigmas de programación destaca la programación orientada a objetos, la cual es, actualmente, el paradigma más extendido para el desarrollo de aplicaciones en general. Entre sus principales características destaca la independencia existente entre sus componentes tanto a nivel de diseño como de implementación lo cual permite realizar diseños altamente modulares y actualizables.

2.5.1. Conceptos

Este paradigma define una serie de conceptos abstractos en función de los cuales se definen todos los componentes del problema, datos y comportamientos, y las relaciones existentes entre ellos:

Clase: El principal elemento conceptual, las clases, son una representación abstracta de un concepto real del problema y engloban tanto su estado, conjunto de información relativa al elemento en un momento dado y su comportamiento. Las clases son elementos con independencia total, a excepción de las herencias, del resto de elementos del diseño e incluso del diseño mismo. Una clase que represente un concepto concreto podrá ser reutilizada tal cual en otra aplicación sin necesidad de recodificarla, simplemente con trasladarla al diseño nuevo.

Permite también, gracias a la ocultación, poder reemplazar un diseño o implementación de una clase por otra que represente el mismo concepto sin necesidad de propagar cambios al resto del diseño o código. Estas características dotan de una gran capacidad de actualización del software desarrollado bajo este paradigma al tiempo que diferencia claramente los elementos conceptuales del diseño, lo que permite a tanto a los desarrolladores como a los clientes comprender fácilmente el diseño del sistema.

Atributos y Métodos: Las clases contienen dos "tipos" de elementos o conceptos: los atributos y los métodos. Ambos son características implícitas de una clase y la principal diferencia entre ambos es que los atributos representan una característica o subcomponente y los métodos un comportamiento o reacción. Se podría decir que los atributos definen "como es" y los métodos "como actúa". Pudiendo ambos a su vez ser definidos como clases el conjunto de clases de un sistema queda relacionado mediante una jerarquía de relaciones "contiene", "forma parte de" o "conoce a" y no existen elementos en el sistema que no estén definidos por una clase.

Herencia: Otro concepto existente es la herencia representando relaciones "es un". La herencia permite definir subclases que, heredando de una o varias clases padres, las especifica o completa añadiendo atributos concretos o métodos específicos. Esto permite realizar diferentes niveles de abstracción a la hora de diseñar un modelo para el problema, permitiendo agrupar elementos diferentes bajo un mismo concepto común para ambos unificando en la clase padre los métodos o atributos comunes a las clases hijo.

Objeto: Siendo las clases y sus relaciones una representación abstracta de conceptos los objetos son la instanciación real de esas clases. Una clase puede tener al mismo tiempo múltiples instancias, objetos, de sí misma, cada una de ellas con un estado o valor de atributos y un estado de comportamiento, métodos en ejecución. Estos objetos se comunicarán entre ellos mediante mensajes, llamadas a los métodos de otros objetos conocidos, según queda definido en el diseño abstracto a nivel de clases. Los objetos son los encargados de representar los elementos reales del problema durante la ejecución de la aplicación, representando así un estado específico del problema y, colaborando entre ellos mediante los mensajes, realizaran la labor destinada a la aplicación.

2.5.2. Características

Algunas de las características de la orientación a objetos más relevantes ya han sido mencionadas anteriormente y definen las directrices generales del diseño orientado a objetos:

Ocultación: La primera de ellas es la ocultación, mediante ella cada clase u objeto está internamente aislada de su entorno. Un objeto es el único responsable de su propio estado y comportamiento, permitiéndose así en un diseño fácilmente divisible, reemplazable e incrementable. Gracias a ello el diseño de una aplicación se convierte en un "puzle" de clases interconectadas entre ellas pudiendo separa el diseño de la representación de los elementos del problema del diseño del comportamiento de la aplicación.

Abstracción: La abstracción permite elevar el nivel de diseño agrupando conceptos que, a priori, puedan parecer diferentes en conceptos más generales que los engloban, permitiendo agrupar bajo una definición de clase diferentes elementos del problema. Una vez definidos estos conceptos se puede ir profundizando en el nivel de diseño especificando esos conceptos en subconceptos, creando así una jerarquía que posteriormente se trasladará al diseño de clases. Esta abstracción permite concretar las características, atributos y métodos, de cada concepto/clase para posteriormente relacionarlos entre sí mediante llamadas, de forma que su comportamiento colectivo realice el trabajo esperado por la aplicación.

Modularidad: Gracias a la independencia entre los diferentes elementos del diseño este se puede subdividir fácilmente en partes más pequeñas, módulos, cada una de las cuales representará de forma independiente una serie de conceptos y comportamientos. Al igual que con las clases, estos módulos pueden distribuirse de forma independiente para ser importados por otras aplicaciones, incorporándolos a su diseño directamente sin necesidad de re implementar los conceptos y comportamientos ya realizados anteriormente. De esta característica se deriva la gran reusabilidad del código generado bajo este paradigma.

Polimorfismo: Esta característica representa la posibilidad de que diferentes conceptos o comportamientos se identifiquen bajo el mismo nombre. Esto dota de una enorme flexibilidad al diseño a nivel de tipado, una clase no necesitará saber con qué otra clase se está comunicando mientras esa clase contenga un método que represente el mismo concepto o funcionalidad esperado y se identifique con el mismo nombre. Esto permite que las clases se relacionen "a ciegas" con otras clases únicamente sabiendo que esas clases son capaces de atender un mensaje dado, es decir, conociendo únicamente una característica conceptual de la misma. De esta forma podremos, por ejemplo, tratar los elementos de una lista heterogénea de una forma homogénea simplemente con que todos sus componentes compartan, nominalmente, los métodos utilizados.

2.5.3. Justificación del paradigma

Como ya se indicó en el capítulo anterior, GEMA debe ser altamente modular y escalable y permitir tanto mejoras internas en un futuro como la adición de módulos que amplíen o complementen su funcionalidad. Se busca obtener un código fácilmente mantenible y actualizable en el que el acoplamiento entre sus elementos sea mínimo de forma que los cambios a realizar sean los mínimos y más localizados posibles.

Dado el alto grado de modularidad alcanzable con este paradigma, la capacidad de conexión y reemplazo de elementos al sistema, la separación de la implementación interna de las clases de su concepción abstracta y, en general, las características anteriormente descritas, hacen de la programación orientada a objetos el paradigma de programación que más se adecúa a los requisitos de ingeniería de la aplicación.

El modelo de diseño en clases, conceptos y atributos relacionados entre sí resulta, a su vez, fácilmente comprensible e independiente de la mayoría de conceptos informáticos y de la programación, facilitando así la comunicación con el cliente durante todas las etapas de desarrollo permitiendo mantener esta comunicación al nivel de abstracción del cliente sin necesidad de hacerle comprender conceptos informáticos de bajo nivel que no le interesan.

2.5.4. Lenguaje de programación: Python

Python [24] es un lenguaje orientado a objetos con cada vez mayor presencia en el mercado. Es un lenguaje interpretado y multiplataforma lo que da una gran flexibilidad a la distribución de aplicaciones desarrolladas en el. Además de incorporar todas las ventajas de la programación orientada a objetos comentadas anteriormente incorpora las suyas propias: es un lenguaje orientado a la reusabilidad, claridad, legibilidad del código y, por extensión, su sencillez.

Estas características desembocan en un lenguaje con el que una vez definido un diseño de objetos este es muy sencillo y rápido de implementar.

Posee tipado dinámico, esto implica que la adecuación de los tipos a la hora de operar se verifica en tiempo de ejecución. Si bien esto permite la aparición de ciertos tipos de errores de programación que difícilmente se dan o se solucionarían fácilmente en lenguajes de tipado estático permite a cambio una gran flexibilidad a la hora de implementar muy importante, aprovechando al máximo las características del polimorfismo y abstracción propias de los objetos.

Consta de reflexión, capacidad de análisis y redefinición del código en tiempo de ejecución. Posee una completa orientación a objetos, todo elemento en Python es un objeto, desde los tipos básicos hasta las funciones, métodos o módulos importados. Esto ofrece la posibilidad de alterar el comportamiento de la aplicación en "caliente" durante la ejecución del programa, tanto por actores externos como por la propia aplicación al analizarse a sí misma.

Es un lenguaje diseñado para ser ampliado y consta de diversos métodos para integrar librerías desarrolladas en otros lenguajes por medio de extensiones o mediante "wrappers" o para integrar Python dentro de aplicaciones desarrolladas en otros lenguajes para utilizarlo, por ejemplo, como sistema de scripting en ellas.

Se puede encontrar una implementación del protocolo SNMP en la librería OpenSource PySNMP [25] para Python. Esta librería resuelve la interfaz de comunicación en red entre los dispositivos y la aplicación de monitorización ofreciendo una capa de programación de alto nivel, lo que permite centrar el diseño en el modelo de objetos y la abstracción de la red a monitorizar sin necesidad de descender al nivel de comunicación en SNMP. Ofrece servicios para el parseo de MIBs y su verificación contra los dispositivos de forma que es posible verificar que variables, descritas en una o varias MIBs dadas, están disponibles en el dispositivo. Esta librería es multiplataforma y da soporte a las versiones 1, 2c y 3 de SNMP tanto bajo IPv4 como IPv6 e importa las librerías necesarias para el tratamiento de tipos ASN.1 utilizados por el protocolo SNMP.

La elección de Python como lenguaje de programación para este proyecto se debe, en primer lugar, a ser un lenguaje extendido en la programación orientada a objetos que incorpora todas las ventajas descritas anteriormente de este paradigma y, en particular, a las características enumeradas anteriormente que permiten cumplir varios de los requisitos de la aplicación:

Soporte SNMP mediante la librería PySNMP que soluciona de manera estándar la comunicación con los dispositivos a monitorizar.

Capacidades de reflexión y de integración que pueden ser explotadas para la adición de scripting capaz de interactuar con todo el sistema o la adición de plugins de forma dinámica.

Disponibilidad multiplataforma que permite migrar fácilmente el proyecto a otros sistemas operativos como Windows.

Su sintaxis sencilla, flexibilidad y capacidades de funcionamiento dinámico, como el tipado, que permiten centrarse en el nivel de diseño sin plantearse limitaciones o restricciones debidas al lenguaje, permitiendo así un diseño de alto nivel independiente del lenguaje de desarrollo.

Existe una amplia variedad de entornos de desarrollo para Python con diferentes prestaciones. Durante el desarrollo de este proyecto se ha decidido utilizar el entorno de programación Eric4 [26]. Este entorno está especialmente diseñado para Python. Dispone de resaltado de sintaxis, autocompletado, sincronización automática con "subversión" [27], consola integrada de sistema y de Python, sistema de 'debugging' y gestión de paquetes de proyecto entre otras funcionalidades.

Capítulo 3

CAPÍTULO 3. MODELO DE COMPONENTES ESCALABLE PARA LA MONITORIZACIÓN

Índice

En este capítulo se expone tanto diseño realizado para la herramienta de monitorización como la estrategia de diseño y desarrollo seguida para su realización e implementación. Se explica, en primer lugar, el modelo de diseño sin entrar en detalles específicos de la aplicación para, sobre este modelo, describir la estrategia seguida. Posteriormente se expondrá, con diferentes niveles de detalle, cada componente según las diferentes capas de diseño realizado y se realizará una exposición de alto nivel conjunta de las mismas. A continuación se analizará cada capa en detalle a diferentes niveles, desde su papel general dentro de la aplicación hasta el detalle de diseño e implementación de cada uno de los módulos y componentes que la forman.

3.1 Modelo y estrategia de diseño y desarrollo

El proceso de diseño se ha realizado según una estrategia mixta top-down y bottom-up. En primer lugar se ha realizado una división de la aplicación en capas de alto nivel a las que se les ha asignado un contexto general de trabajo para, posteriormente, concretar los requisitos de cada una partiendo de los requisitos generales de la aplicación. Se han definido de forma abstracta los interfaces entre las diferentes capas.

El modelo de capas se ha diseñado de forma concéntrica con una capa central que denominaremos núcleo del sistema o “core” rodeada de una serie de capas que incorporan diferentes competencias generales.

Las competencias están normalmente diferenciadas por los diferentes actores con los que trabajará el sistema: núcleo o nivel de monitorización, nivel de red, nivel de presentación o usuarios, nivel de sistema y nivel de procesos, esta última representa las diferentes líneas de ejecución en que se repartirá el trabajo de la aplicación.

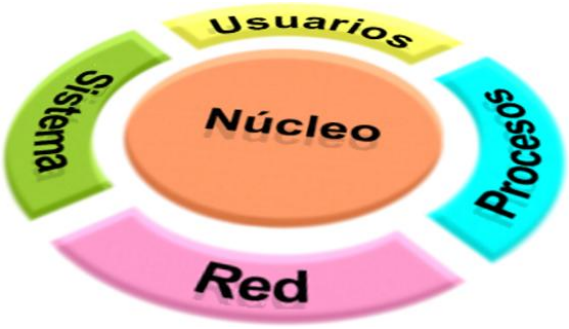


Figura 3.1. Diagrama circular de capas.

Internamente a cada capa el diseño ha sido realizado según una estrategia bottom-up, permitiendo así un modelo de prototipado rápido con el que poder ir consultando al cliente. De esta forma los requisitos y el diseño se van consolidando de manera colaborativa con el mismo al tiempo que los diferentes módulos que conforman cada capa van tomando forma y se van incrementando.

Tras la propuesta de diseño general inicial se desarrolla un primer prototipo centrado en unas pocas funcionalidades o competencias del sistema que se presenta al cliente. En ciclos de, inicialmente, una semana se van presentando nuevas versiones definiendo, mejorando y completando el prototipo en una serie de ciclos de desarrollo y revisión hasta alcanzar un punto en que el prototipo alcanza un grado de madurez suficiente como para poder añadir nuevas funcionalidades y competencias al mismo.

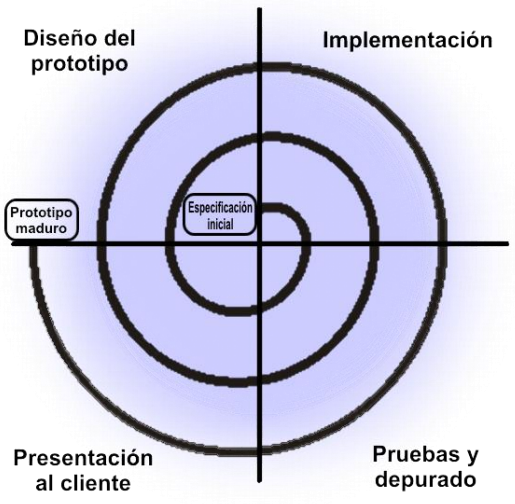


Figura 3.2. Ciclo de desarrollo del prototipo.

En este punto se desarrolla un nuevo prototipo partiendo del anterior como base, añadiéndole nuevas funcionalidades y competencias de forma que, al tiempo que se realizan los ciclos de desarrollo y revisión del mismo, se continúa refinando el prototipo anterior incluido en este. Este proceso se repite una vez alcanzado un nuevo hito de madurez.

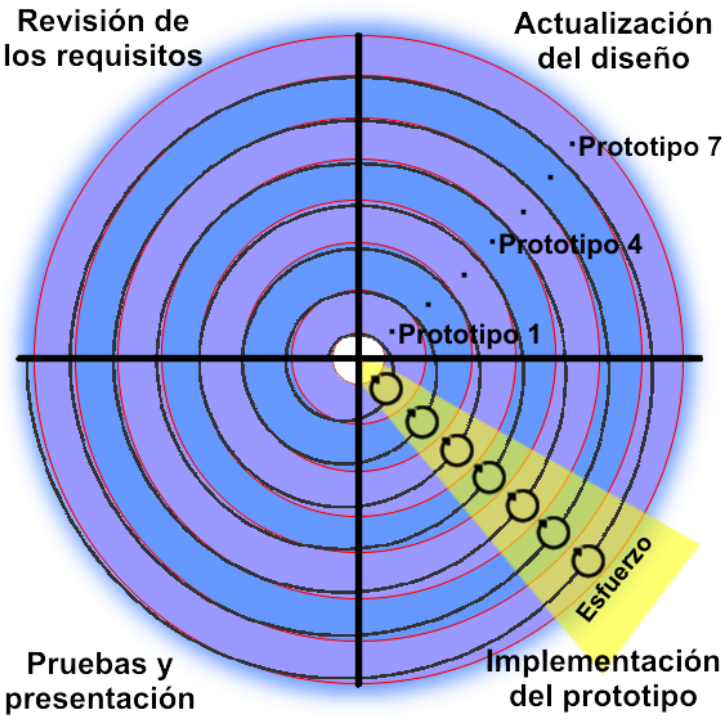


Figura 3.3. Ciclo de desarrollo en espiral por capas de prototipos.

En cada iteración el esfuerzo dedicado se realiza proporcionalmente desde fuera hacia dentro de forma que el mayor esfuerzo se destina a las funcionalidades más recientes o externas y las más antiguas o profundas reciben menos atención. Esto se debe a que, siguiendo este modelo, se asume que las capas internas han sido suficientemente refinadas por lo que necesitan menor atención que las superiores.

El tiempo por ciclo de desarrollo se va dilatando a medida que el prototipo se incrementa y el trabajo a realizar es mayor llegándose a alcanzar ciclos de un mes de duración entre prototipos presentados en las etapas más avanzadas del proyecto.

Tras una serie de prototipos acumulados se alcanza un hito mayor en la madurez del proyecto, en este punto se realiza un ciclo específico sobre el producto generado sin realizar añadidos.

Durante este ciclo el que el esfuerzo de revisión se realiza desde el interior al exterior y de forma más homogénea para consolidar el producto, refinar ajustes entre diferentes capas y corregir posibles errores de diseño generados que se hayan podido propagar por las diferentes iteraciones y que no hayan podido ser atajados en mitad de la rutina normal de desarrollo.

Es en este ciclo de consolidación en el que se realiza la mayor cantidad de refactorización del código. Una vez terminado este ciclo y partiendo del prototipo consolidado se procede a añadir nuevos elementos al mismo y continuar con el modelo de desarrollo hasta que la capa de alto nivel del proyecto queda completada.

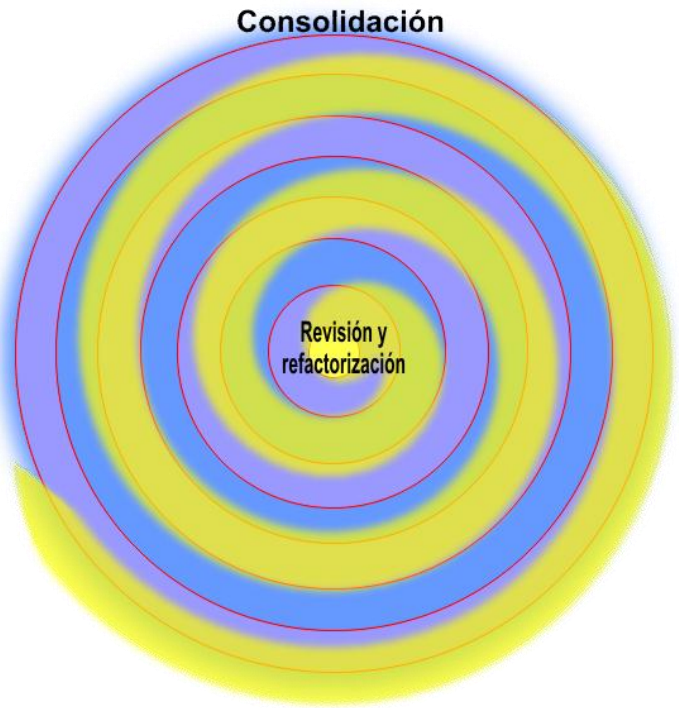


Figura 3.4. Ciclo de consolidación.

Este modelo de desarrollo se aplica internamente en cada capa, una vez que estas alcanzan un nivel de madurez suficiente y se definen los interfaces entre ellas se comienzan a conectar unas con otras. De esta forma las diferentes capas se unifican conformando la aplicación al completo pero manteniendo su independencia interna.

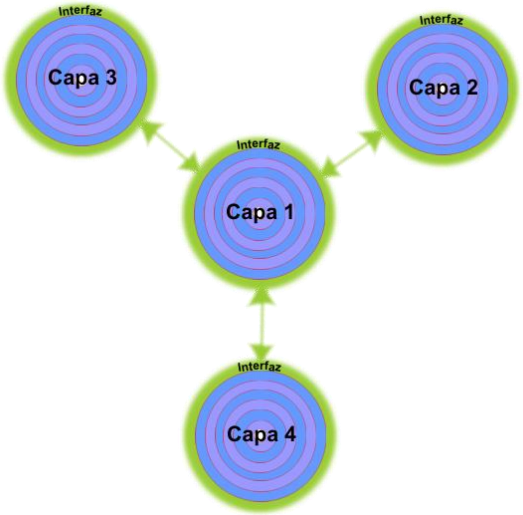


Figura 3.5. Conexionado de capas.

El resultado de esta estrategia de desarrollo es un producto más ajustado a los requisitos del cliente que ha sido desarrollado de una forma ordenada. Esto permite obtener un producto con módulos y capas bien definidas y diferenciadas, favoreciendo así las tareas de depuración y actualizaciones futuras que puedan aparecer.

3.2 División en capas de la aplicación

Esta sección realizará una exposición de alto nivel de las diferentes capas en que se ha dividido la aplicación. Para esta división se ha utilizado un criterio principalmente guiado por los diferentes elementos con los que deberá interactuar cada una. Tras el análisis y diseño inicial se definen cinco capas: Núcleo, Red, Presentación, Sistema y Procesos.

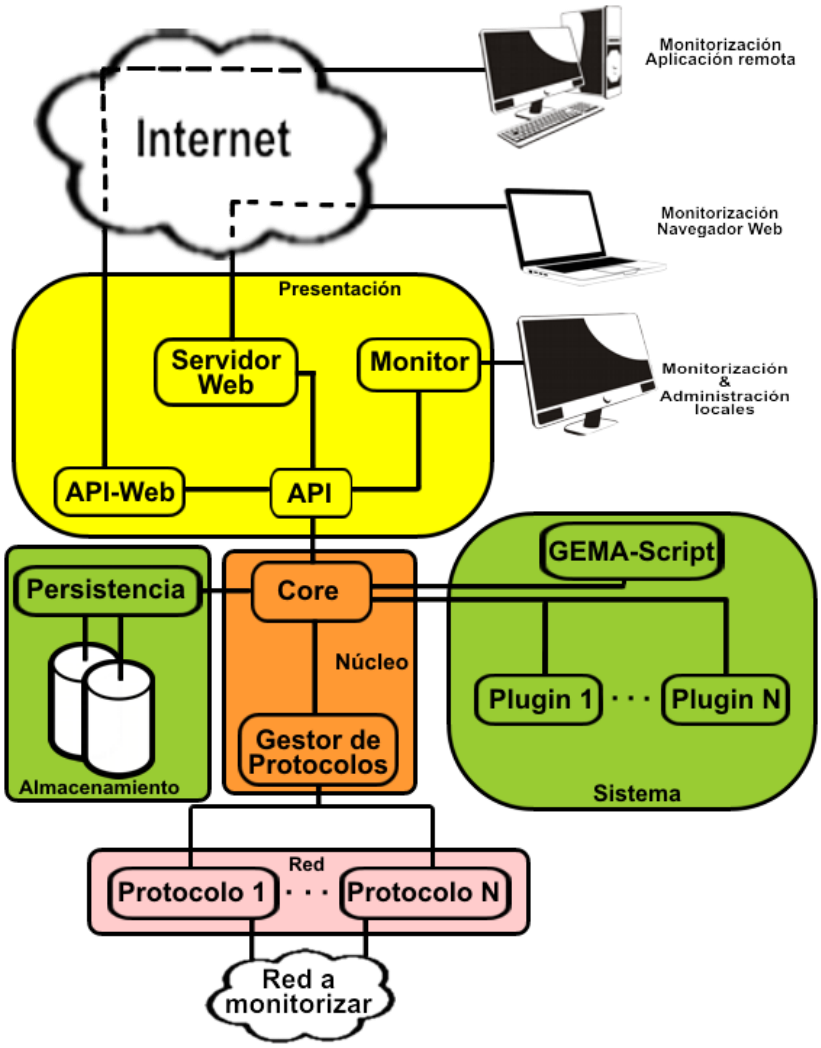


Figura 3.6. Modelo detallado de las capas (exceptuando el nivel de procesos).

A continuación se detallan brevemente las justificaciones, competencias y funcionalidades de cada una de ellas. En los siguientes apartados se ahondará en los aspectos del diseño e implementación de cada una de una forma más exhaustiva.

Núcleo: Esta capa central es la encargada de representar en tiempo real toda la red monitorizada. En ella se engloban los conceptos encargados de la representación y monitorización de todos los elementos de la red: Layout, Nodos, Conexiones, Propiedades y Variables.

Esta capa actúa tanto como centro de información para todo el resto del sistema como de esqueleto central para la interconexión de todos los módulos que componen GEMA.

Red: Esta capa es la encargada de gestionar los diferentes protocolos de monitorización que se deseen implantar en el sistema. Su labor principal es la de actuar como interfaz estándar entre el modelo abstracto de red del núcleo y los diferentes mecanismos de monitorización de redes.

Para su conexión con el núcleo del sistema se implementa un gestor de protocolos de forma que se independiza esta labor del núcleo permitiéndose así incluir y modificar protocolos sin que esto propague ningún cambio al resto de módulos.

Presentación: La capa de presentación engloba tanto los servicios de cara al usuario o administrador como todos los elementos y aplicaciones externas que deseen conectar con GEMA. Esta capa incorpora un API a través del cual cualquier módulo que se desee conectar al sistema se comunicará con el núcleo.

Este API es el único punto de acceso a la capa central, por ello cualquier cambio que se produzca en resto del sistema no se propagará a la implementación de los módulos o plugins conectados. La capa de presentación incluye, inicialmente, un menú de consola que permite al administrador iniciar el resto de servicios y plugins implementados.

Sistema: En la capa de sistema y extensiones se localizan, principalmente, aquellos módulos encargados de interactuar con el sistema en el que se esté ejecutando la aplicación. Entre estos módulos se encuentran los servicios de escucha de logs y traps snmp y los sistemas de persistencia que se implementen.

En esta capa se localizarán también aquellos servicios que, dado su carácter crítico, no deban ser accedidos desde el exterior a través del API, entre ellos el módulo encargado de analizar el lenguaje de scripting Gema-Script.

Procesos: La capa de procesos incluye todos aquellos módulos encargados de controlar las diferentes tareas de monitorización y ejecución de propiedades. Estos elementos se implementan y desarrollan en una capa diferenciada para separar las competencias relativas a la representación de la información del flujo de operaciones sobre ella. Esto cobra especial importancia en el momento de considerar y diseñar los diferentes hilos de ejecución o "threads" concurrentes de los que consta GEMA.

De este modo es posible modificar los mecanismos de control de concurrencia y técnicas de paralelización que sean necesarias para la optimización de la ejecución de la aplicación en diferentes sistemas base sin que esto interfiera con el modelo de datos y diseño del resto de componentes.

3.3 Diseño e implementación del núcleo

El núcleo de GEMA es el encargado de representar de forma abstracta y homogénea los diferentes elementos de la red a monitorizar. Para ello se definen una serie de clases que permiten representar de forma estructurada toda la jerarquía de la red tanto a nivel lógico como físico.

Este nivel conecta también con el interfaz del API de la capa de presentación de forma que desde esa capa superior no sea visible ningún otro elemento de GEMA, controlando así todos los accesos y operaciones que se realicen desde el exterior.

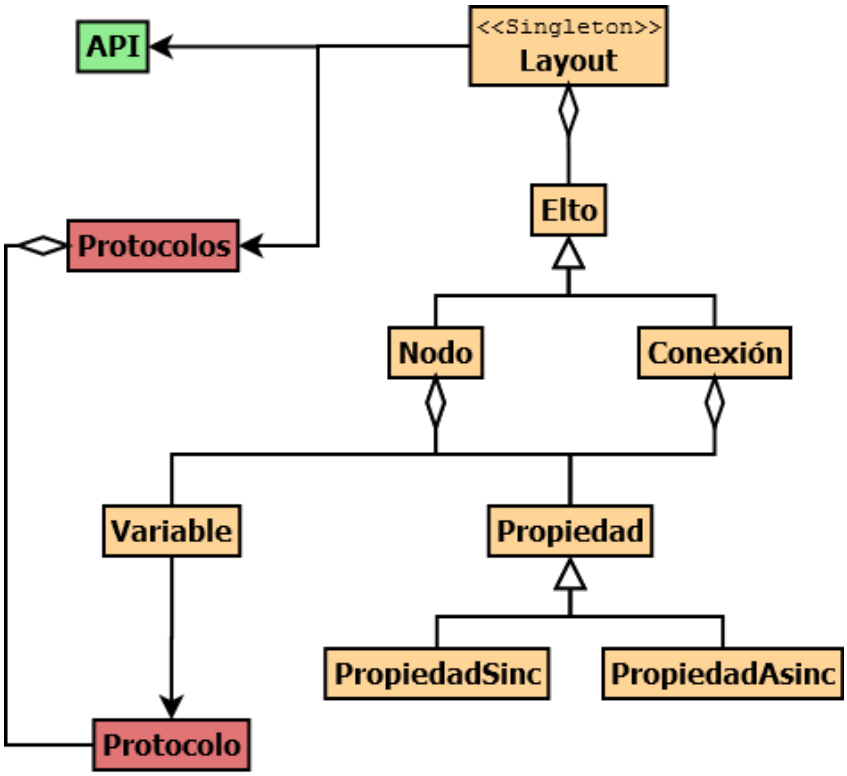


Figura 3.7. Modelo simplificado de clases del núcleo.

Esta capa se conectará con la capa de red a través del gestor de protocolos implementado. Este gestor será referenciado directamente desde el Layout de forma que sea accesible por los elementos que lo necesiten.

Todos los componentes adicionales, módulos y plugins conectados a GEMA directamente accederán al Layout y por extensión a todo el modelo de red e información monitorizada a través de la Api o directamente a través del singleton del Layout y por ello deberán ser convenientemente verificados antes de añadirlos y activarlos dentro del lanzador de la aplicación.

Los interfaces con el exterior, sin embargo, deberán acceder necesariamente a través de la Api, por ello no será necesario verificarlos pues el propio Api se encargará de controlar y verificar la corrección de las operaciones solicitadas al tiempo que, mediante un módulo de gestión de permisos de usuario, regula quien tiene acceso a qué elementos u operaciones del sistema.

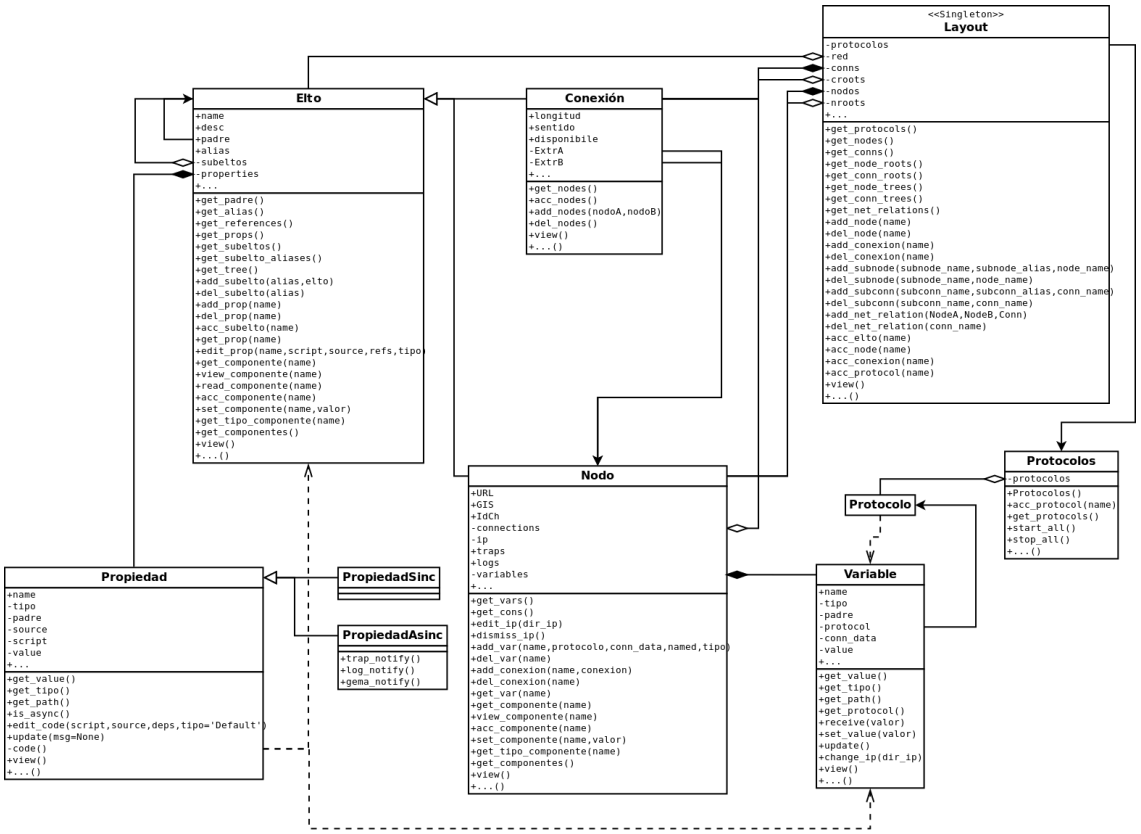


Figura 3.8. Diagrama de clases del núcleo reducido.

3.3.1 Layout

El Layout representa la totalidad de los elementos de la red a monitorizar así como sus relaciones, incluye también una referencia al gestor de protocolos permitiendo así un acceso abstracto a estos desde este nivel de forma que puedan ser referenciados desde el resto del sistema.

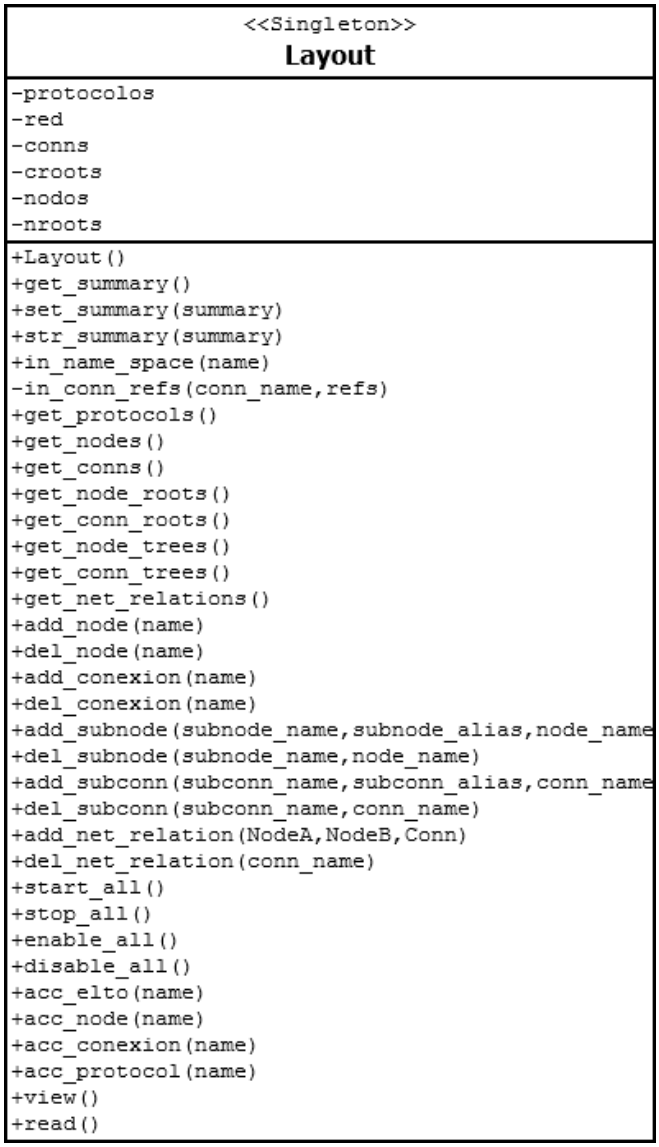


Figura 3.9. Clase Layout.

La representación de la red se realizará mediante una serie de estructuras englobadas en la clase Layout:

Dos poliárboles de elementos de red, uno para los nodos o elementos monitorizables y otro para las conexiones entre los mismos.

La estructura de árbol permite definir relaciones de agregación y composición tanto lógicas como físicas. Es posible así representar un armario que contiene diferentes multiplexores, los distintos elementos de un multiplexor, agrupaciones de conexiones e incluso canales lógicos dentro de una misma fibra.

La existencia de varios árboles en la estructura, poliárbol, permite definir elementos o jerarquía totalmente independientes entre ellas.

La implementación de estas estructuras se realiza mediante referencias a los ascendentes y descendientes de cada elemento en la clase correspondiente, nodo o conexión, la cual contendrá los métodos necesarios para realizar los recorridos correspondientes.

El Layout almacenará una lista actualizada de los elementos raíz de cada árbol así como una lista completa de todos los elementos del sistema para permitir el acceso directo a los mismos.

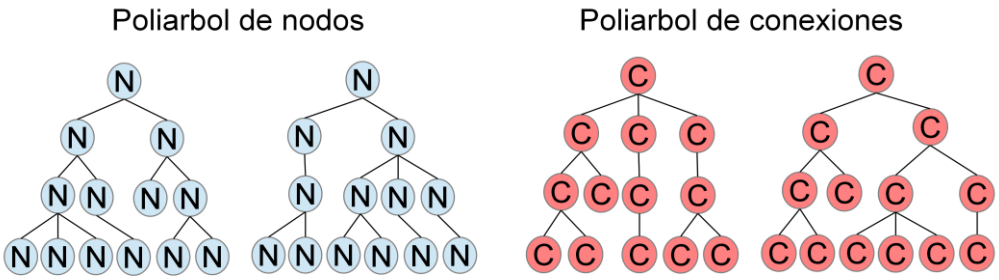


Figura 3.10. Representación de los poliárboles de elementos.

El Layout consta también de un grafo que representa las uniones entre nodos mediante las conexiones, de esta forma se representa la topología de la red a monitorizar, esta representación no tiene por qué ser estrictamente de la topología física o lógica de la red si no que es posible combinar ambas.

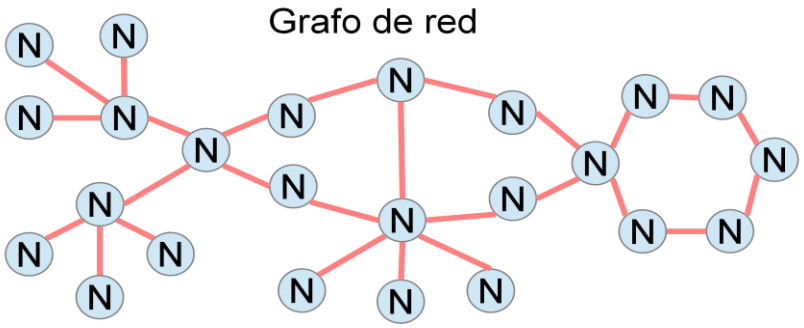


Figura 3.11. Representación del grafo de red.

La implementación de estas estructuras se realiza mediante diccionarios de Python, tablas hash, lo que permite tanto el acceso directo a los elementos por su identificador, nombre, como el acceso secuencial a los mismos.

Nótese que los diccionarios contienen referencias a los objetos, de esta forma el mismo objeto puede estar referenciado en varios diccionarios sin ser duplicado.

En el caso del grafo de red este se implementa como un diccionario de diccionarios representando así una tabla de adyacencias entre nodos con un diccionario de conexiones en cada campo.

A pesar de comprobarse que esta tabla será siempre triangular se decide implementarla de forma completa para optimizar el acceso bidireccional a su información, no obstante se tiene en cuenta esta propiedad a la hora de recorrerla.

Nodo / Nodo	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5
Nodo 1	**N / A**		**Conn 1, Conn 7**		
Nodo 2		**N / A**		**Conn 2**	**Conn 3, Conn 5**
Nodo 3	**Conn 1, Conn 7**		**N / A**		**Conn 4**
Nodo 4		**Conn 2**		**N / A**	
Nodo 5		**Conn 3, Conn 5**	**Conn 4**		**N / A**

Figura 3.12. Implementación en tablas de la red.

Los diferentes métodos implementados permiten añadir, modificar o eliminar elementos de las estructuras así como generar listas de los mismos tanto en formato plano como anidadas según la jerarquía configurada.

Serializado de elementos:

$$[name, name, name, \dots]$$

Serializado de arboles:

$$[root\_name\ [name\ [\dots], name\ [\dots]\ ], root\_\ name\ [name\ [\dots], name\ [\dots]\ ]\ ]$$

Serializado del grafo:

$$[\ (node\_name, [\ (conn\_name, node\_name\ ), (conn\_name, node\_name\ ) \dots]\ ) \dots\ ]$$

Figura 3.13. Formato serializado de los listados del Layout.

Consta de accesos directos a los elementos mediante su identificador gracias a las funciones “acc_” de forma que los módulos que los utilicen puedan mandar los mensajes correspondientes a los mismos sin necesidad de que el Layout haga de intermediario en todas las operaciones. Esta funcionalidad es principalmente utilizada por el código Python generado para las propiedades.

Se implementan métodos que permitan enviar las órdenes de arranque y parada a todos los elementos de la red evitando a los módulos externos realizar los correspondientes recorridos de las jerarquías.

El método “view” devolverá una vista completa del Layout en una estructura de lista de forma que nuevamente se abstrae su implementación interna del resto de módulos que lo accedan.

Finalmente se implementan métodos dedicados a la serialización de la clase permitiendo así abstraer su diseño de los mecanismos de persistencia que se implementen.

El Layout al completo se diseña según un patrón singleton, de esta forma todos los elementos de la capa que deseen acceder a la estructura abstracta de red podrán hacerlo en cualquier momento sin necesidad de conocer previamente al Layout.

Para la implementación del patrón singleton el lenguaje no posee directamente un mecanismo que implemente este patrón de diseño, sin embargo se aprovecha la orientación a objetos completa del lenguaje, Python, a nivel de módulo. En Python cada módulo importado es, en sí mismo, un objeto en memoria e internamente el sistema de "imports" gestiona los módulos de forma que cada importación no crea una nueva instancia. De esta forma cada módulo importado por el sistema en diferentes partes será el mismo en todas ellas.

La solución implementada consiste en crear un módulo abstracto (no pertenece al diseño) con una variable a nivel de módulo que apunte al Layout junto con una serie de métodos que operen sobre ella. De esta forma cada vez que se importe este módulo abstracto desde cualquier punto del código se asegura que es el mismo y su instancia es única.

Accediendo a través de él a la variable definida se consigue asegurar que todos los mensajes llegan a la misma instancia de Layout sin necesidad de almacenar una referencia al mismo en cada modulo implementado. A través de las funciones implementadas se permitirá reiniciar el Layout actual así como cargar en el singleton uno ya construido como, por ejemplo, uno recuperado por un modulo de persistencia. También permite solicitar la destrucción del Layout de forma que pueda restaurarse el estado inicial de a aplicación en cualquier momento de forma ordenada.

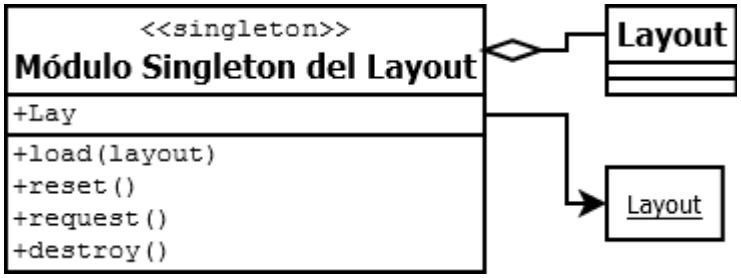


Figura 3.14. Solución de implementación del singleton.

Este módulo abstracto actuará al mismo tiempo de patrón factoría, siendo capaz de proporcionar nuevos layouts desvinculados del singleton si se necesitasen mediante la función "request". Para realizar los direccionamientos bastará con hacerlo a través del módulo abstracto importado:

import SingleLay as SL

SL.Lay.Método_del_Layout

3.3.2 Elemento de red

Los nodos y las conexiones heredan de una superclase común denominada elemento de red, Elto, que agrupa la mayor parte del comportamiento de estos objetos.

Los elementos de red son objetos capaces de contener propiedades y que incluyen una serie de atributos descriptivos básicos.

También disponen de todos los elementos necesarios para el control y ejecución de las propiedades que incluyen. Todo elemento de red puede, a su vez, contener un número indeterminado de elementos de red así como una referencia a su antecesor, conformado de esta forma las estructuras de árbol que se incluyen en el Layout.

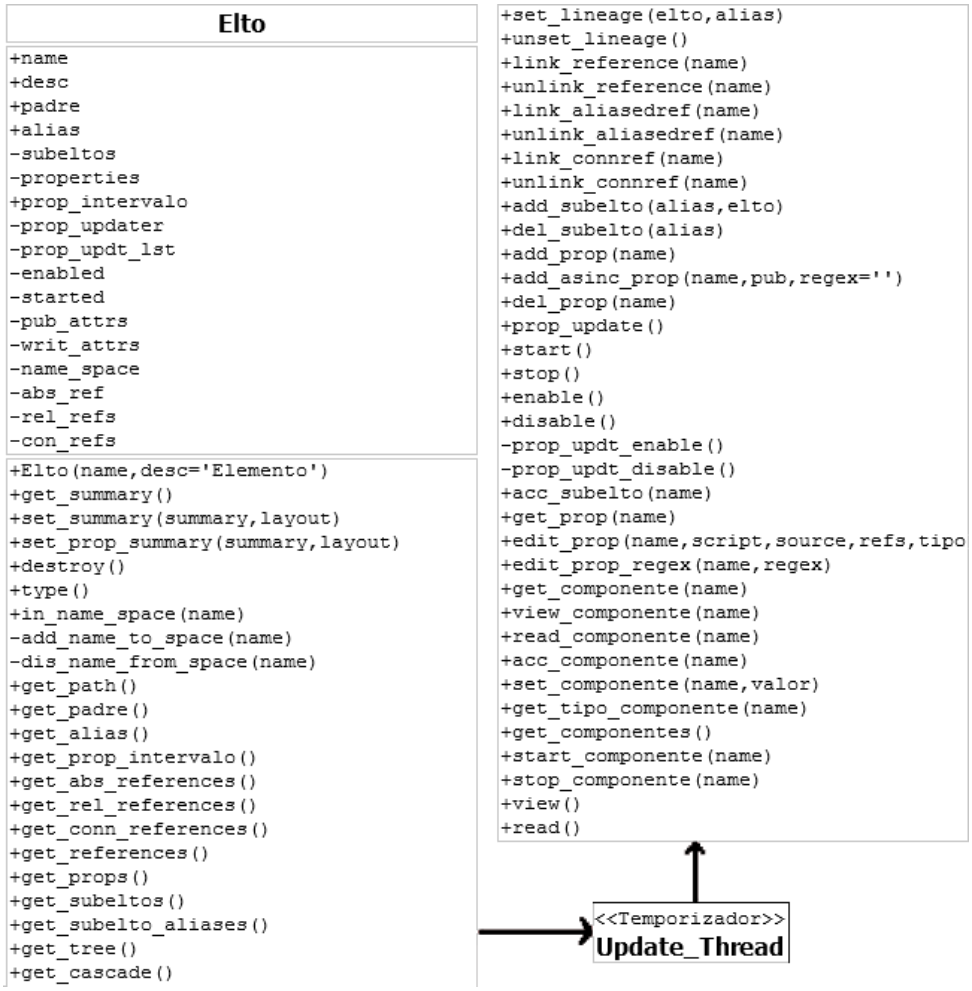


Figura 3.15. Clase Elto.

Cada elemento de red posee un Updater, clase encargada de activar una función iterativamente con un intervalo de tiempo variable. Este Updater es el encargado de ordenar la actualización de las propiedades activas en cada momento. Para ello se definen dos diccionarios.

El primero contiene todas las propiedades incluidas en el elemento y el segundo únicamente aquellas que han de ser actualizadas. Será este último diccionario sobre el que se operará cada vez que el Updater solicite una actualización de las propiedades. Existe una función encargada de notificar el intervalo actual de actualización, de forma que el valor de este sea completamente externo al Updater.

Incluye una serie de métodos para solicitar la activación y desactivación de todas sus propiedades y Updater evitando así que la implementación de estas deba ser conocida por los módulos externos.

A diferencia del Layout, Elto proporciona métodos específicos para operar sobre sus componentes: Propiedades, Updater, etc. Evitando así propagar el conocimiento de su diseño e implementación internos.

Finalmente incorpora métodos de vista y serialización de su información para su consulta y almacenamiento por un sistema externo de persistencia.

3.3.3 Nodo y Conexión

Los nodos extienden la clase elemento de red aportándole la capacidad de contener variables monitorizables, permitiendo de esta forma que los elementos definidos se comuniquen con los equipos a monitorizar. También le dotan de la capacidad de recibir un número indeterminado de conexiones y de una serie de atributos descriptivos propios de los nodos junto con los métodos necesarios para gestionarlos.

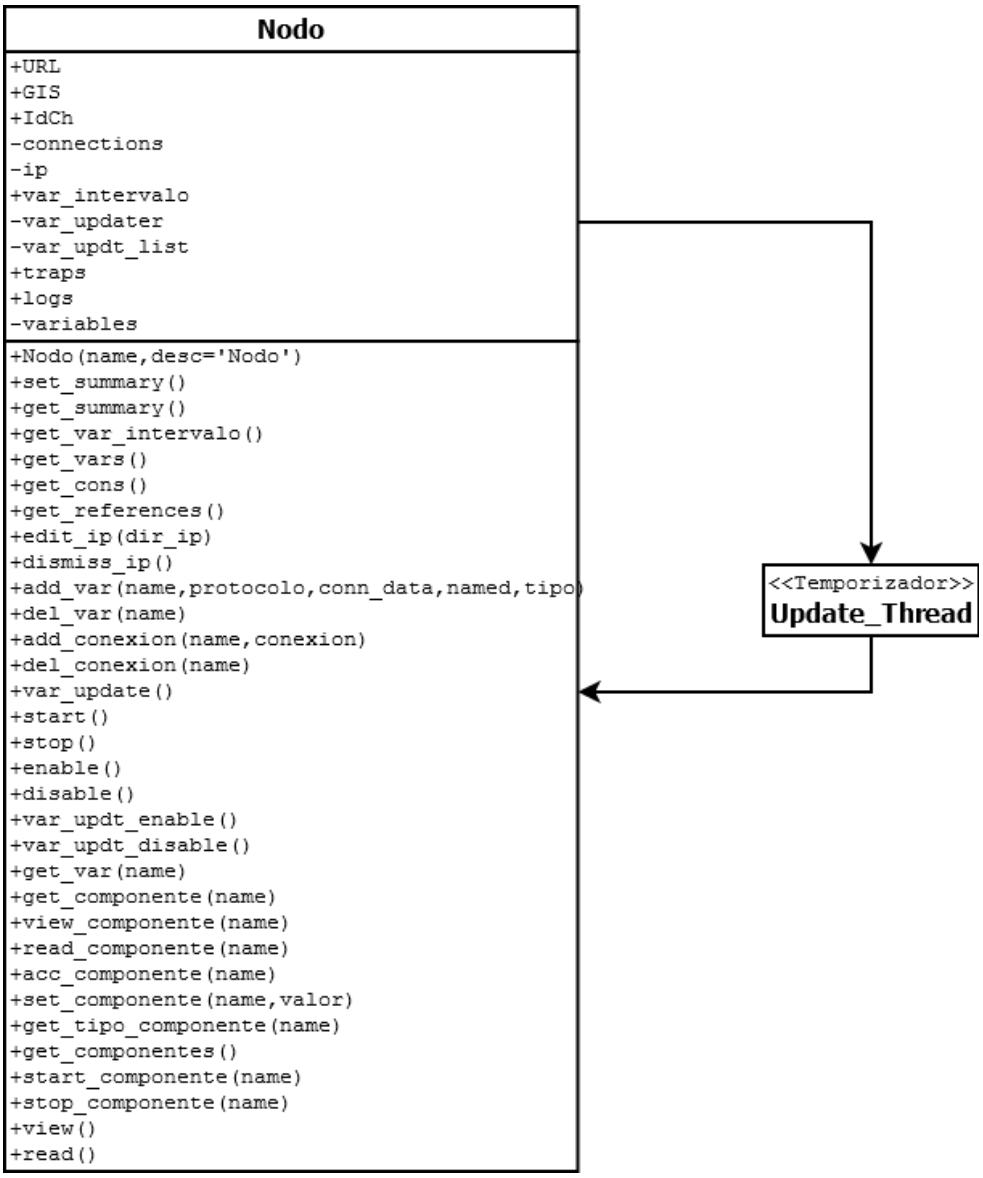


Figura 3.16. Clase Nodo.

La mayor parte de los métodos reimplementados en esta clase son extensiones al método básico de la clase Elto para poder operar sobre las variables sin necesidad de utilizar métodos específicos para ellas.

Para su implementación se referenciará, en el punto oportuno del código, al método de la clase padre, evitando así replicar código y reduciendo la aparición de errores debidos a ello.

Para la gestión de la actualización de las variables la clase Nodo incorpora un Updater dedicado así como los métodos requeridos por el Updater para realizar su trabajo de forma análoga al diseño utilizado con las propiedades en la clase Elto.

Las conexiones extienden la clase Elto permitiéndola interconectar a otros dos elementos, Extremo A y extremo B, permitiendo así representar las fibras, colecciones de fibras y canales dentro del sistema a monitorizar.

Al igual que en la clase Nodo la mayor parte de los métodos implementados extienden los métodos de la clase padre sin duplicar el código de los mismos. Incluye también los métodos necesarios para gestionar su relación con los nodos a conectar.

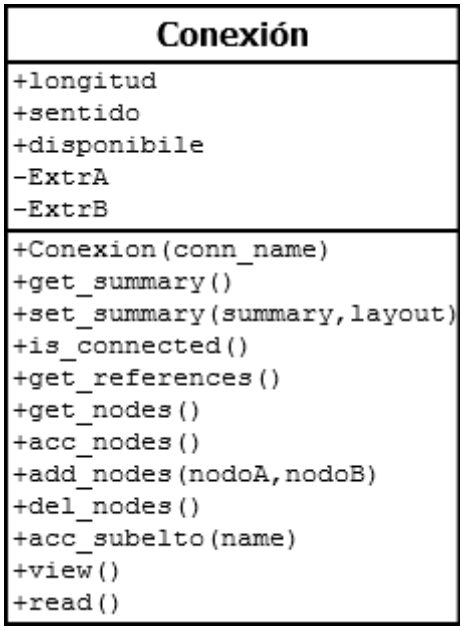


Figura 3.17. Clase Conexión.

Tanto la clase Nodo como la clase Conexión extienden los métodos de acceso a la vista y de serialización de la clase padre Elto para incorporar la información específica de cada una de ellas.

3.3.4 Variable

Las variables representan un valor accesible en un dispositivo físico conectado a la red. Este valor es accedido mediante un protocolo concreto de forma totalmente transparente a la variable.

La clase incluye una serie de atributos descriptivos así como una referencia a su Nodo padre. Posee, además, un conjunto de referencias a la misma desde las propiedades.

Este conjunto ayudará a la hora de determinar si la eliminación de un elemento concreto del Layout podría dañar la integridad referencial del mismo evitando hacer un costoso análisis de las relaciones en el momento de la modificación.

Posee también dos atributos importantes relacionados con la comunicación con el dispositivo. En primer lugar una referencia al objeto Protocolo que será responsable de ejecutar la comunicación y que deberá respetar un estándar definido para ello. En segundo lugar los datos necesarios para realizar esta comunicación almacenados en el Conn_Data.

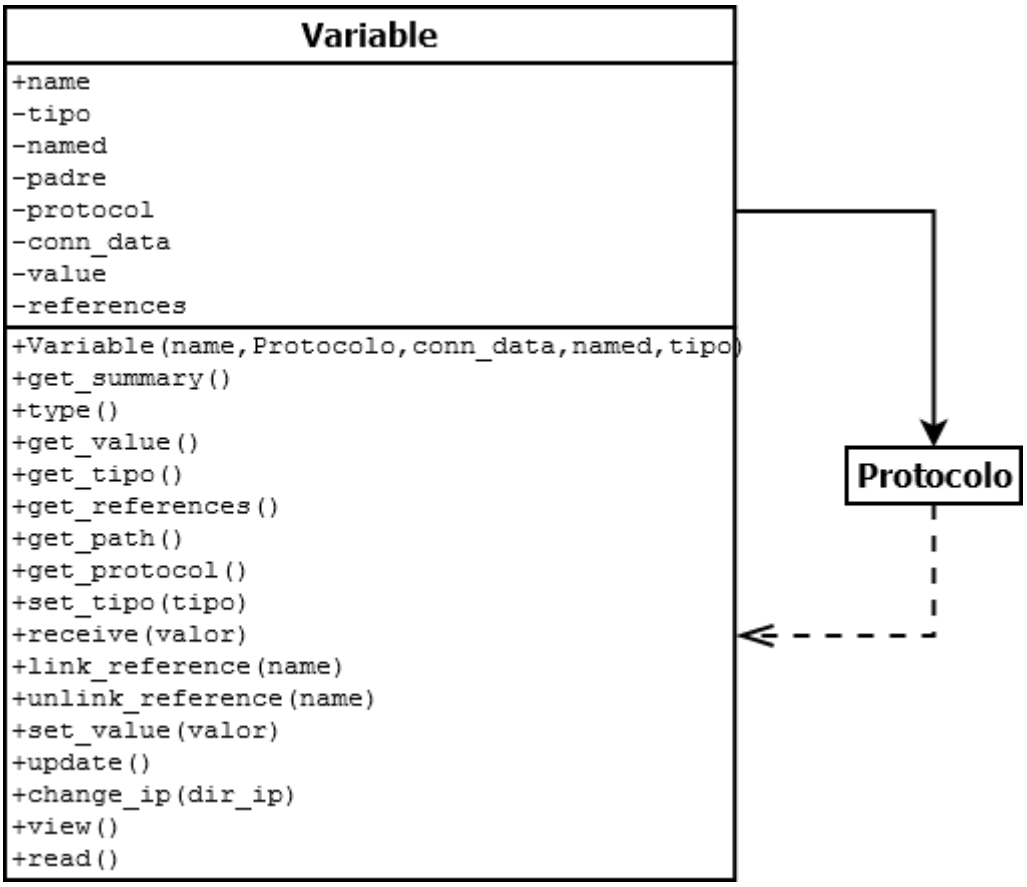


Figura 3.18. Clase Variable.

El Conn_Data es una caja negra dentro del sistema y únicamente el protocolo que lo ha generado será capaz de interpretarlo. Cuando la variable requiera de una operación sobre el protocolo le proporcionará a este el Conn_Data a modo de llave o tarjeta de identificación.

Con este Conn_Data el protocolo deberá ser capaz de identificar de forma precisa la variable dentro de la red, comunicarse con el dispositivo para ejecutar la operación solicitada y devolver el resultado tratado adecuadamente a la variable que lo solicitó. Deberá ser también capaz de proporcionar un nuevo Conn_Data tanto en el instante de la creación de la variable como en el momento en que algún dato relativo a la conexión de la red se modifique.

De esta forma GEMA es completamente insensible al funcionamiento interno de los protocolos, quedando estos reducidos a unas simples operaciones de escritura, lectura y actualización del Conn_Data.

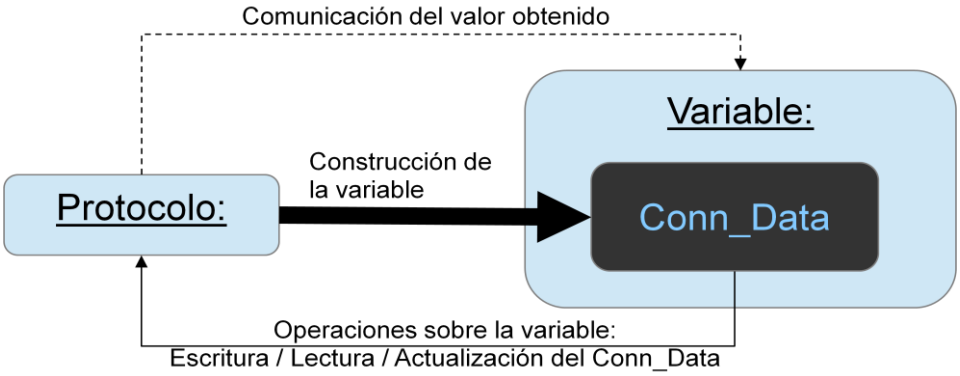


Figura 3.19. Creación y uso del Conn_Data.

El Conn_Data se obtiene del protocolo durante el proceso de creación de la variable. Durante este proceso el protocolo tomará un papel activo para solicitar los datos necesarios para su creación. Este proceso será explicado en detalle en la sección dedicada a los protocolos.

La operación de actualización de una variable si divide en dos pasos. En primer lugar la variable solicita la actualización de su valor al protocolo cuando su método “update” es invocado. Posteriormente el protocolo proporcionará a la variable su nuevo valor mediante el método “receive".

Esto permite implementar un mecanismo asíncrono de actualización de forma que ningún proceso de GEMA quede esperando a que la solicitud de red sea resuelta. Sin embargo es responsabilidad del Protocolo liberar la ejecución de la llamada “get_value” realizada por el método “update” antes de realizar la petición de red.

Si esta condición no se cumpliese la actualización sería síncrona, con los perjuicios que eso conlleva cuando estamos en un escenario de comunicación en redes de gran envergadura y grandes distancias sujetas a posibles averías o incidencias (bloqueos de llamadas, grandes tiempos de espera debido a la latencia, etc.)

La operación de escritura de una variable se realiza mediante el método “set_value”. Es responsabilidad nuevamente del Protocolo implementar internamente está petición de forma asíncrona para evitar bloqueos o esperas innecesarias.

Las variables implementan dos métodos de vista, “view” y “read” el primero es una descripción completa de la variable, el segundo contendrá únicamente la información relevante de monitorización, esto es así para poder reducir el flujo de datos durante la monitorización al mínimo necesario.

De la misma forma que el resto de clases, Variable tiene su propio mecanismo de serialización para su persistencia, sin embargo no dispone de un método de carga.

Para la carga de una variable esta deberá ser creada de nuevo, esto es debido a que el Conn_Data no puede ser serializado. El Conn_Data no puede ser serializado y reconstruido debía a, en primer lugar, que es una caja negra y debería ser competencia del Protocolo.

En segundo lugar porque podría ser dependiente del momento de su creación (sistema de claves dinámicas, asignación de identificadores dinámico, etc.) y por lo tanto ha de ser regenerado en el proceso de creación de la variable.

3.3.5 Propiedad

Las propiedades son el mecanismo mediante el cual GEMA adquiere conocimiento sobre las labores de monitorización y administración de redes. Mediante las propiedades es posible tratar la información de monitorización, atender eventos, inferir nueva información e incluso tomar decisiones y actuar sobre el sistema.

Básicamente una propiedad es un fragmento de código que será ejecutado en unas condiciones concretas. Por lo general está condición es un cronómetro implementado por el Updater de su Elemento padre, sin embargo otras podrán ser activadas únicamente ante la llegada de un evento. En función de esta característica se definen dos subclases que heredan de la clase Propiedad: Propiedad Síncrona y Propiedad Asíncrona.

La propiedad contiene una serie de atributos descriptivos básicos como son su nombre, su tipo, su valor actual y el elemento padre al que pertenece. Junto con ellos se incorporan los métodos necesarios para leerlos o actualizarlos.

Incluye dos atributos específicos relativos a su método de llamada: “Pub” que nos indicará la categoría de mensajes a los que ha de responder o “Sync” en caso de ser síncrona y el atributo “Regex” que especifica una expresión regular que habrá de cumplirse por el mensaje que recibe como parámetro para poder ejecutarse. Estos dos atributos permiten controlar adecuadamente el momento de ejecución de la propiedad evitando gasto de proceso innecesario.

Al igual que las Variables la propiedad posee un conjunto de las referencias a si misma desde otras propiedades para facilitar el cálculo de la integridad referencial dentro de GEMA.

Finalmente consta de una serie de atributos relativos al código que ha de ejecutar. El primero de ellos es el código “Script” que la define. Este código es almacenado de forma informativa y no es procesado de forma alguna por la propiedad. Esto permite al sistema de scripting que se implemente recuperar el código con que se configuró una propiedad en concreto.

El segundo atributo es el “Source”, este atributo contiene el código Python generado por el sistema de scripting y que deberá ser ejecutado por la propiedad Este código es la definición de una función que realizará la labor que se haya implementado en el script.

Esta definición será ejecuta durante el proceso de edición de la propiedad de forma que en tiempo de ejecución quedará definido un nuevo método dentro de la propiedad. Esto es posible gracias al tipado dinámico y a la introspección de Python. Esta función definida se llamará “code”.

Los atributos “AbsDeps”, “RelDeps” y “ConDeps” incluyen información sobre las referencias a otros elementos del sistema, dependencias, que aparecen dentro del código de la propiedad.

En ellos se almacenan las referencias directas a los objetos y su función principal es comunicar a estos elementos que están siendo referenciados para que actualicen sus conjuntos de referencias. Este proceso se realiza mediante los métodos "Link" en caso de alta de la propiedad y "Unlink" en caso de baja.

El flag "Stable" evitará la ejecución del código de la propiedad si este se ha detectado como incorrecto o ha generado fallos en la ejecución.

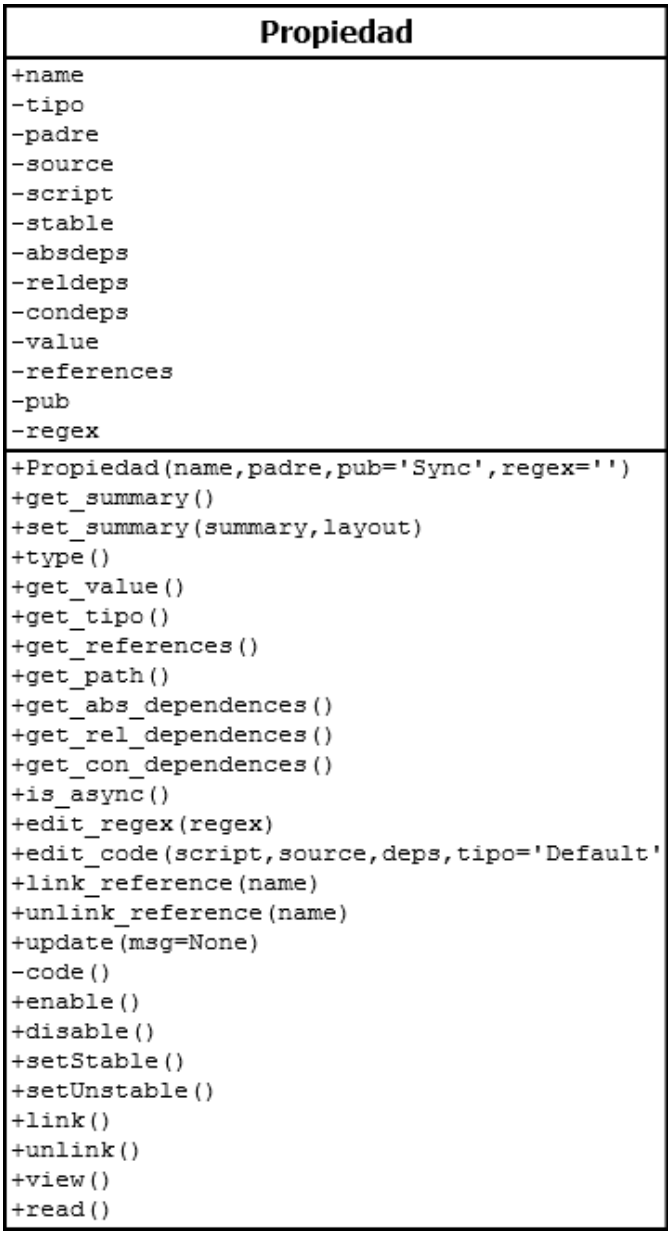


Figura 3.20. Clase Propiedad.

El atributo "Code" es el verdadero "cerebro" de la propiedad. En Python no existe una diferencia real entre atributo y método. Todo son objetos referenciados, incluso las funciones son objetos, de forma que la única diferencia entre un atributo y un método es que estos últimos son objetos ejecutables.

En este atributo, inicializado a 'None', será redefinido en tiempo de ejecución cuando se ejecute la definición de la misma almacenada en "Source".

En el momento que esta definición es ejecutada y si no se producen errores en este proceso quedará redefinido el atributo “Code” pasando a ser ahora un objeto función ejecutable que recibirá como parámetro un mensaje indeterminado. Esta nueva función es la que será invocada al solicitarse un update de la propiedad.

Este mecanismo, apoyado en la capacidad de Python de redefinir su código en tiempo de ejecución, nos permitirá modificar el comportamiento de las propiedades y por extensión de GEMA, sin necesidad de reiniciar el sistema.

Como ya se introdujo anteriormente, existen dos subclases de propiedades, síncronas y asíncronas. Será de estas dos subclases de las que se instancien todos los objetos, dejando la clase Propiedad como una clase madre abstracta de la que heredarán todas los demás tipos de propiedades. De ella heredarán tanto las dos definidas como las que pudieran necesitarse en un futuro, por ejemplo, propiedades programadas para una fecha y hora específicas.

Propiedades Síncronas: Son propiedades que se ejecutarán cada cierto intervalo de tiempo definido por su elemento padre. Recibirán, a priori, un mensaje nulo en todas sus llamadas.

Son, en todos los aspectos, contenidas en la definición de la clase madre Propiedad y su declaración como clase hija no extiende ningún método ni aporta ningún atributo a la clase madre.

Su definición como clase se debe tanto a respetar el diseño impuesto de clase madre abstracta como a permitir y facilitar el enriquecimiento de la clase con las características que en un futuro pudieran ser necesarias.

Propiedades Asíncronas: Estas propiedades serán invocadas cuando un mensaje de la categoría a la que se haya subscrito, Trap, Log o Gema-msg (mensajes generados internamente por otras propiedades), cumpla con su expresión regular.

La intención de esta subclase es implementar un comportamiento de respuesta automático, ya sea de actuación como de aviso, ante un evento del sistema de forma completamente asíncrona.

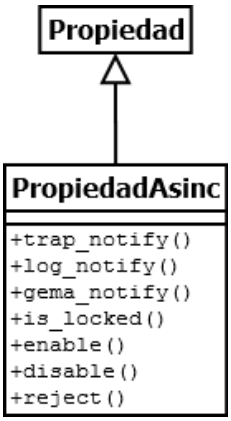


Figura 3.21. Clase Propiedad Asíncrona.

Estas propiedades recibirán como parámetro el mensaje que las ha activado. Su implementación incorpora a la implementación general de la clase madre únicamente los métodos necesarios para ser utilizadas por el sistema de control de subscripciones de GEMA que se expondrá más adelante.

El proceso de ejecución de las propiedades se puede observar en el siguiente diagrama:

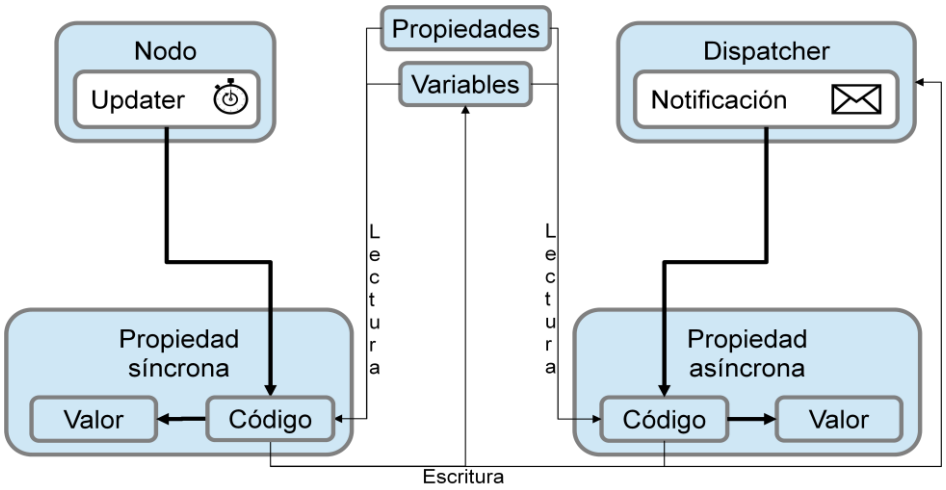


Figura 3.22. Proceso de ejecución de las propiedades.

En primer lugar la propiedad es despertada por el thread del updater en caso de ser síncrona o por un thread específico creado por el dispatcher en caso de ser asíncrona.

Una vez activada, la propiedad ejecutará su código, accediendo a los valores de cualquier variable o propiedad definida en el sistema e incluso atributos de los elementos de red. Las propiedades asíncronas recibirán, además, el mensaje que las despertó como parámetro en su llamada.

En función de estos valores y el código definido en ellas podrán ordenar escrituras en las variables de los diferentes dispositivos de red y generar nuevos mensajes internos que serán enviados a sus subscriptores e incluso podrán despertar nuevas propiedades asíncronas.

De esta forma se pueden crear sistemas complejos de reacción ante diferentes incidencias en la red monitorizada utilizando una jerarquía modular de propiedades, comunicar la información pertinente al exterior mediante mensajes e incluso podría llegar a comunicar dos instancias de GEMA mediante la subscripción de una a la otra.

Una vez terminada la ejecución del código la propiedad almacenará un valor resultado que podrá ser accedido tanto por otras propiedades como por las interfaces gráficas al usuario mostrando el estado de los distintos elementos u operaciones realizadas.

3.4 Diseño e implementación de la capa de red

La capa de red incluye todos los protocolos de red que podrán ser utilizados por la aplicación. Los protocolos habrán de ser implementados manteniendo una interfaz mínima común, de esta forma se podrán dar de alta dentro de GEMA sin necesidad de modificar más que un fichero de configuración que los liste. Incluye además un gestor de protocolos que, en base al fichero de configuración, los mantiene accesibles.

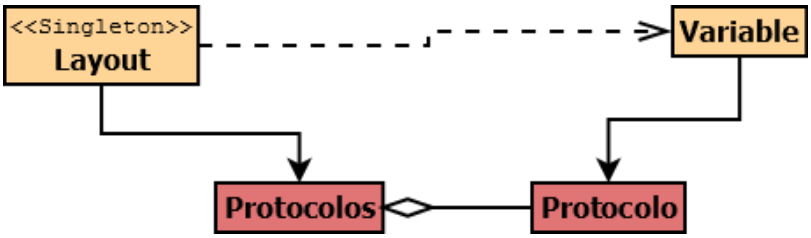


Figura 3.23. Modelo simplificado de clases de la capa de red.

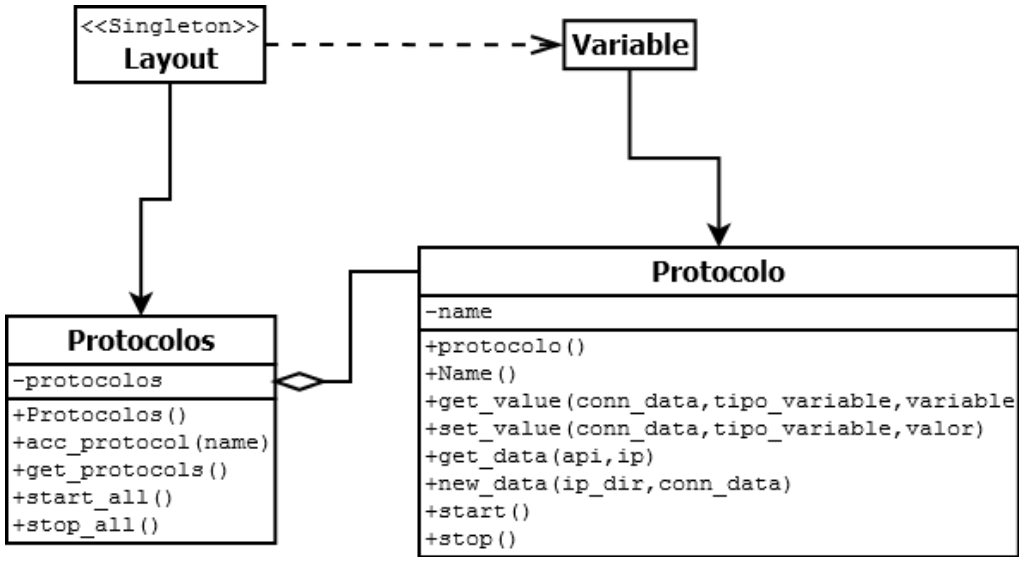


Figura 3.24. Diagrama de clases del nivel de red reducido.

3.4.1 Gestor de protocolos

El gestor de protocolos mantiene indexados todos los protocolos dados de alta y permite al sistema acceder a este índice y obtener acceso a cada uno de ellos.

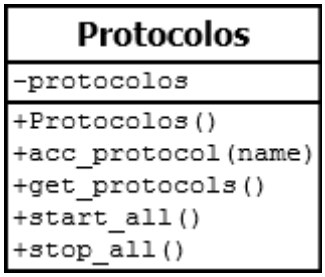


Figura 3.25. Clase del Gestor de Protocolos.

Los protocolos son añadidos de forma dinámica durante la ejecución de la aplicación. Para la implementación de este comportamiento se utiliza el "magic method" de Python "__import__". En Python se denomina "magic method" a una serie de métodos especiales que permiten realizar ciertas operaciones que no suelen estar disponibles de forma directa. Estos métodos son en gran parte los responsables de dotar a Python de su capacidad de introspección.

En el caso del método "__import__" permite ejecutar cómo método la labor de la directiva "import" del lenguaje. De esta forma podemos importar un módulo desde una sentencia más del programa. Debe tenerse en cuenta que antes de esta sentencia el módulo no estará importado y por lo tanto cualquier referencia al mismo provocará un fallo de ejecución.

La solución de implementación consiste en realizar un MAP con este método sobre la lista de protocolos incluida en el fichero de configuración almacenando al mismo tiempo su resultado en la variable "modulos".

Una vez realizado esto todos los protocolos habrán sido importados y estarán referenciados como objetos dentro de la variable "modulos". Tras ser importados se recorre de nuevo la lista obtenida del fichero y se realiza la instanciación de la clase "clase_protocolo" definida en cada uno de ellos almacenándose en un diccionario denominado protocolos utilizando su nombre como campo clave.

El identificador de clase "clase_protocolo" se define como identificador estándar para cualquier módulo que implemente un protocolo. Quedando de esta forma estandarizado el constructor del objeto Protocolo pudiéndose instanciar de forma transparente y homogénea todos ellos.

Este proceso se realiza durante la construcción del objeto gestor de protocolos, sin embargo sería posible realizarlo en cualquier momento durante la ejecución para añadir un protocolo "en caliente".

No será posible darlo de baja de esta manera, si bien, una vez comprobado que no está siendo utilizado por ninguna variable, el protocolo podría ser eliminado del diccionario de protocolos accesibles por el gestor. Sin embargo cualquier hilo de ejecución o simplemente el espacio que ocupa en memoria seguiría consumiendo recursos.

3.4.2 Protocolo

Los protocolos han de respetar una interfaz mínima con el sistema de forma que su alta en el mismo no requiera de cambios ni en la clase Variable ni en el gestor de protocolos.

Para ello habrán de constar de un componente, Conn_Data, que incluirá, a modo de caja negra, toda la información específica del mismo. El Conn_Data será almacenado por las variables y utilizado como identificador ante el protocolo a la hora de solicitar las operaciones de red requeridas.

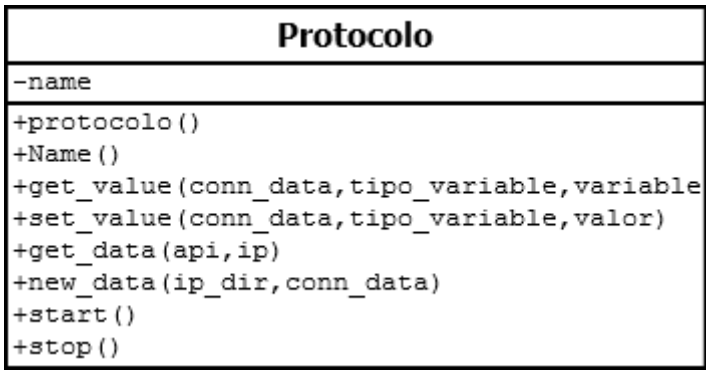


Figura 3.26. Clase Protocolo genérica.

En la imagen observamos los componentes mínimos que debe contener un protocolo. En primer lugar como atributo un nombre único de protocolo que lo identifique unívocamente. A continuación una serie de métodos que conforman el interfaz estándar utilizado por GEMA para comunicarse con el protocolo. Esta es la visión de un protocolo desde la aplicación y cualquier otra funcionalidad o comportamiento especial estará oculto al sistema en métodos internos, que podrán se accedidos, de ser necesario, por módulos especiales o plugins específicamente diseñados para trabajar con ese protocolo.

A continuación realizamos una breve descripción de los diferentes métodos que componen este interfaz estándar:

Name: Devolverá el nombre del protocolo que será su identificador único dentro del sistema.

get_value: Es el método encargado de realizar la monitorización activa de una variable. Recibe como parámetro un Conn_Data del que se extraerán los datos de conexión específicos del protocolo. Con estos datos se realizará una operación GET sobre el dispositivo de la red y su valor será comunicado mediante una llamada al método “receive” de la Variable indicada en la llamada.

El parámetro “Tipo_Variable” será utilizado para realizar las traducciones o interpretaciones necesarias entre el dispositivo y GEMA, de forma que cualquier diferencia de formato sea totalmente transparente para ambos.

Si bien no es un requisito, se recomienda encarecidamente que la implementación de este método sea asíncrona. Para ello podremos dividirla en dos partes, de forma que la ejecución de este método termine inmediatamente tras anotar la solicitud. Evitaremos así bloquear el proceso que realizó la llamada durante el tiempo de latencia de red.

set_value: Mediante este método se permite una gestión activa de los dispositivos de la red mediante la escritura de sus diferentes variables. Recibe, al igual que el caso anterior, un Conn_Data con el que solicitar una operación SET sobre el dispositivo con el Valor proporcionado en sus parámetros.

Al igual que en el método anterior, el parámetro indicando el tipo de la variable se proporciona para las operaciones de traducción que sean necesarias. Es responsabilidad del protocolo revisar y adecuar el valor dado al tipo y formato requerido por el dispositivo.

Al igual que en el caso anterior se recomienda encarecidamente una implementación asíncrona de este método.

new_data: Permite renovar la información de conexión de una variable. Recibirá el Conn_Data actual junto con los nuevos datos de red de su dispositivo. Con esta información generará un nuevo Conn_Data que será entregado a la variable para permitirla identificarse en sus posteriores solicitudes.

get_data: Este método se invocará durante el alta de una variable. Recibirá un conector a un API de interfaz gráfica que será utilizado directamente por el protocolo para solicitar los datos que considere necesarios para la construcción del Conn_Data.

Devolverá una lista de definiciones de variable cada una de las cuales incluirá, además del Conn_Data, una serie de valores necesarios para su construcción: Una referencia al propio protocolo, un diccionario de literales para traducir los valores, su tipo y su nombre.

Debido a la heterogeneidad de los posibles protocolos a utilizar se implementa un mecanismo genérico para obtener los Conn_Data de cada uno.

Durante el alta de una variable de un protocolo concreto será este el que tome el control del interfaz externo que esté solicitando la operación mediante un API proporcionado en la llamada.

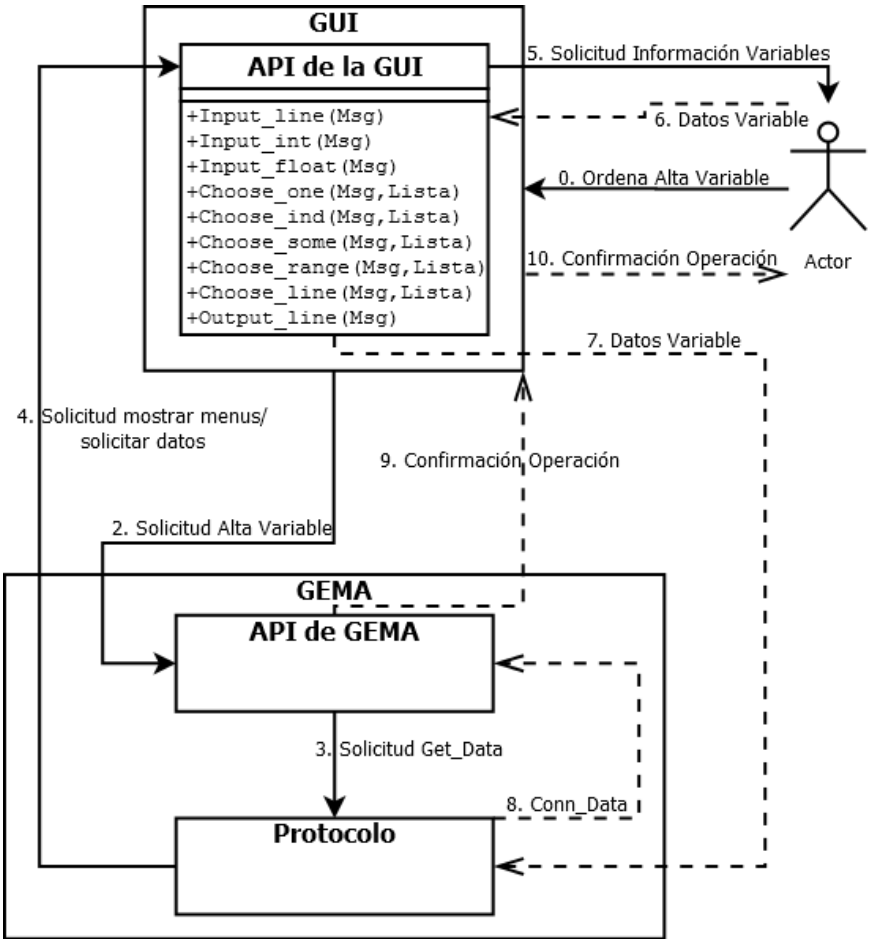


Figura 3.27. Diagrama de secuencia para el alta de variables.

Como se muestra en el diagrama, el proceso de alta de una variable es iniciado por una petición por parte del usuario. Ante esta solicitud el interfaz solicitará a GEMA que inicie el proceso de alta de variable poniendo a su disposición un API en la GUI encargado de mostrar al usuario las solicitudes que GEMA considere oportunas.

Desde el punto de vista del interfaz no hay conocimiento del diseño o estructura interna de GEMA, no distinguiendo que módulo de la misma es el que realiza las solicitudes. De esta forma la implementación de los interfaces es totalmente independiente de los protocolos existentes.

Una vez recibida la solicitud, el API de GEMA traslada la petición al protocolo correspondiente informándole del API de la GUI a utilizar.

Con esta información el protocolo comenzará a solicitar, a través del interfaz, los datos necesarios para el alta de la variable al usuario. Una vez obtenidos generará el Conn_Data correspondiente y se lo devolverá a GEMA para la creación de la variable.

De esta forma se crea una independencia entre GEMA y el protocolo, el proceso de creación de los Conn_Data y mecanismos internos del protocolo son transparentes a GEMA. Igualmente la gestión y modelado del la variable dentro de GEMA es totalmente indiferente a nivel del protocolo.

La lista de funcionalidades incluidas en el API de la GUI permite, mediante operaciones simples, realizar la petición de los datos necesarios de los protocolos. Permite la solicitud de datos simples mediante las llamadas 'input', selecciones y selecciones múltiples mediante las llamadas 'choose' y finalmente la notificación de mensajes mediante la función 'output'.

Siguiendo este diseño, el añadido de protocolos a GEMA no implicará ningún cambio de diseño en el sistema. Igualmente cualquier cambio en el núcleo no afectará a los protocolos ya implementados permitiendo así ser reutilizados, incluso, por otras aplicaciones. Más adelante podremos estudiar una implementación específica de este mecanismo para el protocolo SNMP en el siguiente apartado.

Por debajo de todos estos métodos requeridos de forma estándar el protocolo podrá implementar, internamente, todos aquellos que considere necesarios para su correcto funcionamiento.

Del mismo modo el contenido del Conn_Data es completa responsabilidad del protocolo, pudiendo llegarse incluso a implementar un thread de ejecución dentro del mismo para, por ejemplo, un sistema de actualización dinámico de claves. Sin embargo la implementación del protocolo deberá ser consecuente con el coste de proceso y memoria que conllevaría realizar este tipo de implementaciones.

3.4.3 Ejemplo de Implementación de un protocolo: SNMPv1

A continuación se hace una breve descripción del diseño e implementación de la clase para el protocolo SNMPv1 incluido de serie en GEMA. Esta implementación puede servir de ejemplo a la implementación de otros protocolos a incluir en el sistema. Mostrará también un caso práctico de implementación y uso del API de las GUI para proceder al alta de una variable en este protocolo.

Hace uso de un thread interno, importando la clase Updater ya introducida anteriormente, para crear un modelo asíncrono de respuesta a las peticiones de lectura de las variables. También gestiona una serie de ficheros de definiciones, MIBs, propias del protocolo así como las librerías de definición de tipos, ASN1, que importa junto con la librería PySNMP de Python.

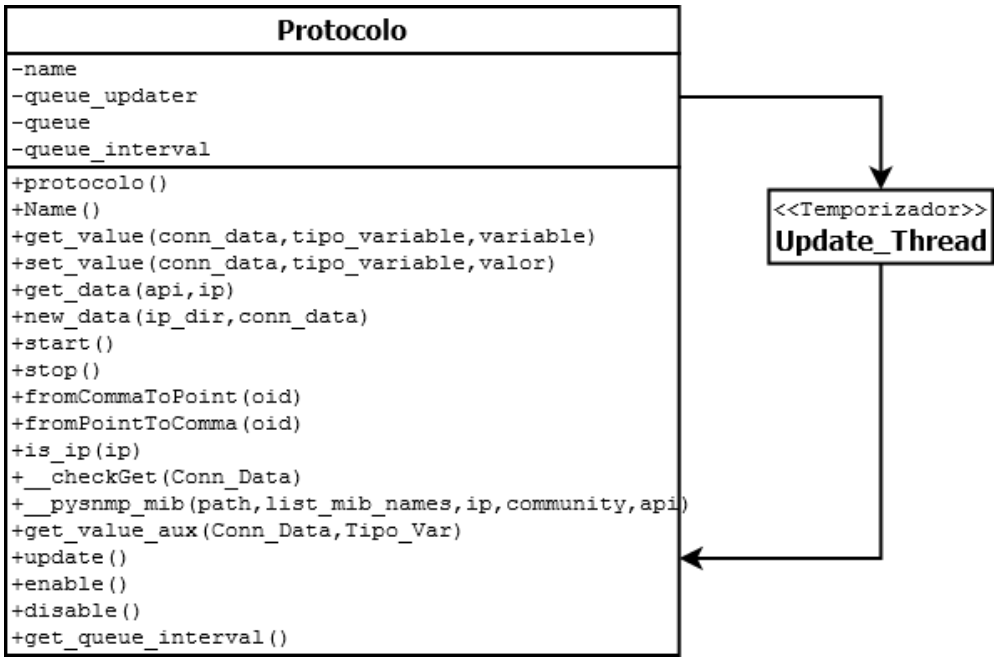


Figura 3.28. Clase Protocolo SNMPv1.

Para la implementación del protocolo SNMPv1 nos hemos apoyado en la librería PySNMP de Python que incorpora todas las funcionalidades de nivel de red y comunicación requeridas por el protocolo. Para ello se han añadido una serie de métodos internos a los requeridos por el interfaz estándar ya expuesto.

En primer lugar podemos observar tres métodos dedicados a la verificación y transformación de los tipos de datos:

- fromCommaToPoint: Traslada los OID de un formato separado por comas a uno separado por puntos.
- fromPointToComma: Traslada los OID de un formato separado por puntos a uno separado por comas.
- is_ip: Verifica que el parámetro pasado es una dirección IP correctamente construida en versión 4.

La funcionalidad de lectura de valores utilizada para la monitorización se implementa de forma asíncrona mediante el uso de un buffer interno en el que se encolan las peticiones. Este buffer es recorrido por un thread propio, instancia de la clase Updater, para ejecutar las peticiones necesarias de forma independiente a las solicitudes por parte de las variables.

De esta forma se permite, además, realizar las optimizaciones a las peticiones que el protocolo considere necesarias. Por ejemplo, en la implementación realizada, a la hora de encolar una petición se comprueba que esa misma variable real no esté ya en la lista.

Si la variable estuviera ya solicitada simplemente se añadiría un solicitante a una lista adjunta. De esta forma se realizaría una única solicitud SNMP y se comunicaría el resultado a todos los solicitantes indicados. De esta forma la redundancia en las definiciones dentro de GEMA no afecta al volumen del tráfico de red generado.

El método ‘get_value’ invocado por la variable almacenará la solicitud en el buffer ‘queue’. Esta solicitud incluirá el Conn_Data proporcionado, el tipo de la variable, necesario para las traducciones y una referencia a la variable sobre la que se ejecutará la función de retorno ‘receive’ una vez obtenido el valor. Cada elemento en el buffer es una terna con el primer y segundo elemento fijos al Conn_Data y tipo de la variable, el tercer elemento de la terna es un conjunto de referencias a variables que lo han solicitado.

Esta implementación evitará realizar peticiones SNMP redundantes en caso de que la misma variable física sea solicitada por diferentes variables definidas dentro de GEMA. Permitiendo optimizar así el tráfico de red generado por la monitorización.

En función del intervalo definido, ‘queue_interval’, el ‘queue_updater’, solicitará la ejecución del método update, que recorrerá el buffer por sus Conn_Data e invocando al método ‘get_value_aux’ que será el encargado de realizar la petición SNMP GET sobre el dispositivo con los datos especificados en el Conn_Data. Una vez obtenido el valor este será notificado a todas las variables en el conjunto de solicitudes mediante la invocación del método ‘receive’.

El método ‘get_value_aux’ se encarga así mismo de traducir los valores que sea necesario entre el tipo utilizado por el dispositivo y el tipo esperado indicado por GEMA. Estos tipos son directamente dependientes de la definición de las MIBs y GEMA no hace uso ni tratamiento de ellos en forma alguna.

Los métodos ‘enable’ y ‘disable’ se encargan de gestionar la activación y destrucción del Updater para permitir un cierre ordenado de todos los procesos de GEMA.

‘set_value’ es el método encargado de ordenar la escritura de la variable, su implementación en este caso es directa y no asíncrona, invocando los métodos de la librería PySNMP para realizar la petición SET sobre el dispositivo. Al igual que ‘get_value_aux’ realizará las traducciones de tipos que sean necesarias.

‘new_data’ es el encargado de renovar los Conn_Data. Recibirá como parámetros el anterior Conn_Data, compuesto por el OID de la variable, la IP del dispositivo, el community utilizado y el tipo de la variable. Devolverá el Conn_Data con el campo IP actualizado. Manteniendo así al ocultación de su estructura interna ante el resto de GEMA.

Finalmente ‘get_data’ es el encargado de obtener los datos necesarios para el alta de una variable y de generar los Conn_Data que serán proporcionados a GEMA para su posterior identificación. El proceso seguido es la especificación a los requisitos de SNMP del mecanismo presentado en el anterior apartado.

A continuación se realiza una breve descripción del mismo en el contexto de SNMP. En los diagramas de secuencia mostrados se muestran todos los elementos involucrados en el alta de una variable: Actor o usuario, Interfaz o GUI, API de la GUI, el nodo lógico en el layout al que se añadirá la variable, GEMA, el API de GEMA, el protocolo involucrado y, finalmente, el dispositivo físico a monitorizar, un CMUX4+.

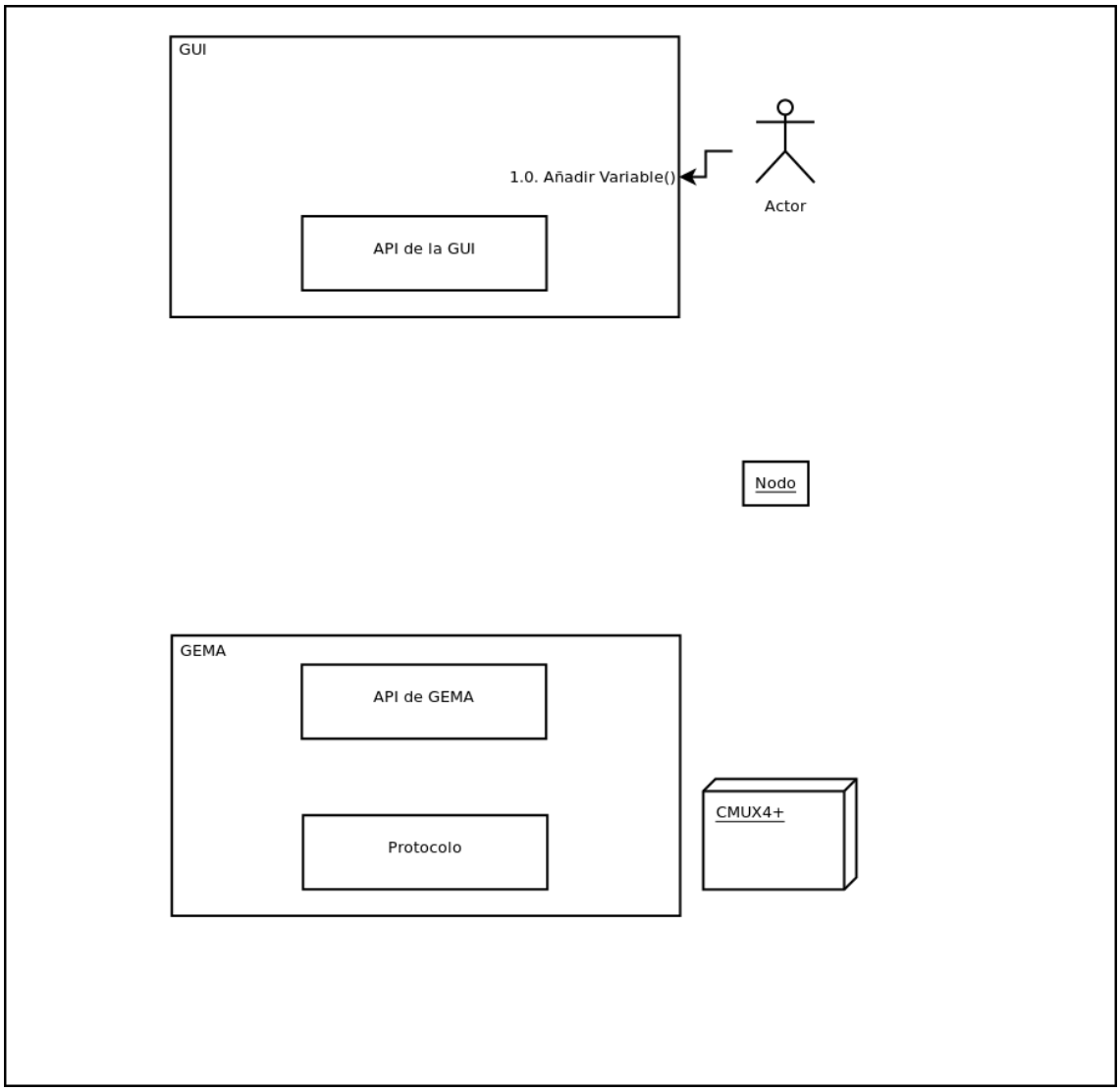


Figura 3.29. Alta de variable SNMPv1 Fase 1.

1.0. El proceso comienza con una solicitud de alta de variable por parte del usuario.

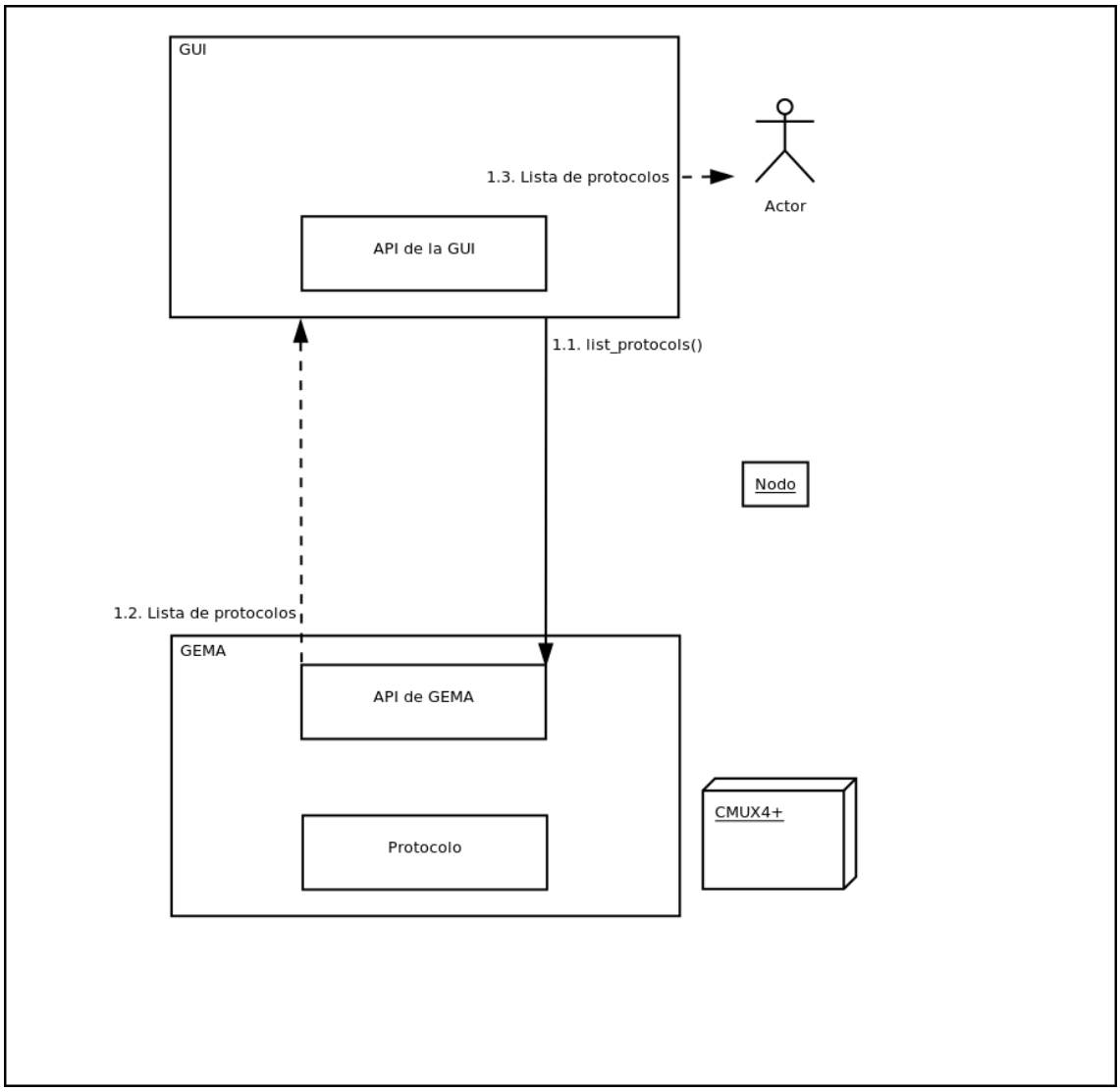


Figura 3.30. Alta de variable SNMPv1 Fase 2.

1.1. El interfaz solicita a GEMA la lista de protocolos disponibles para iniciar el proceso de alta.

1.2. Tras la solicitud por parte del interfaz GEMA proporcionará la lista de protocolos disponibles al interfaz para que el usuario seleccione el indicado.

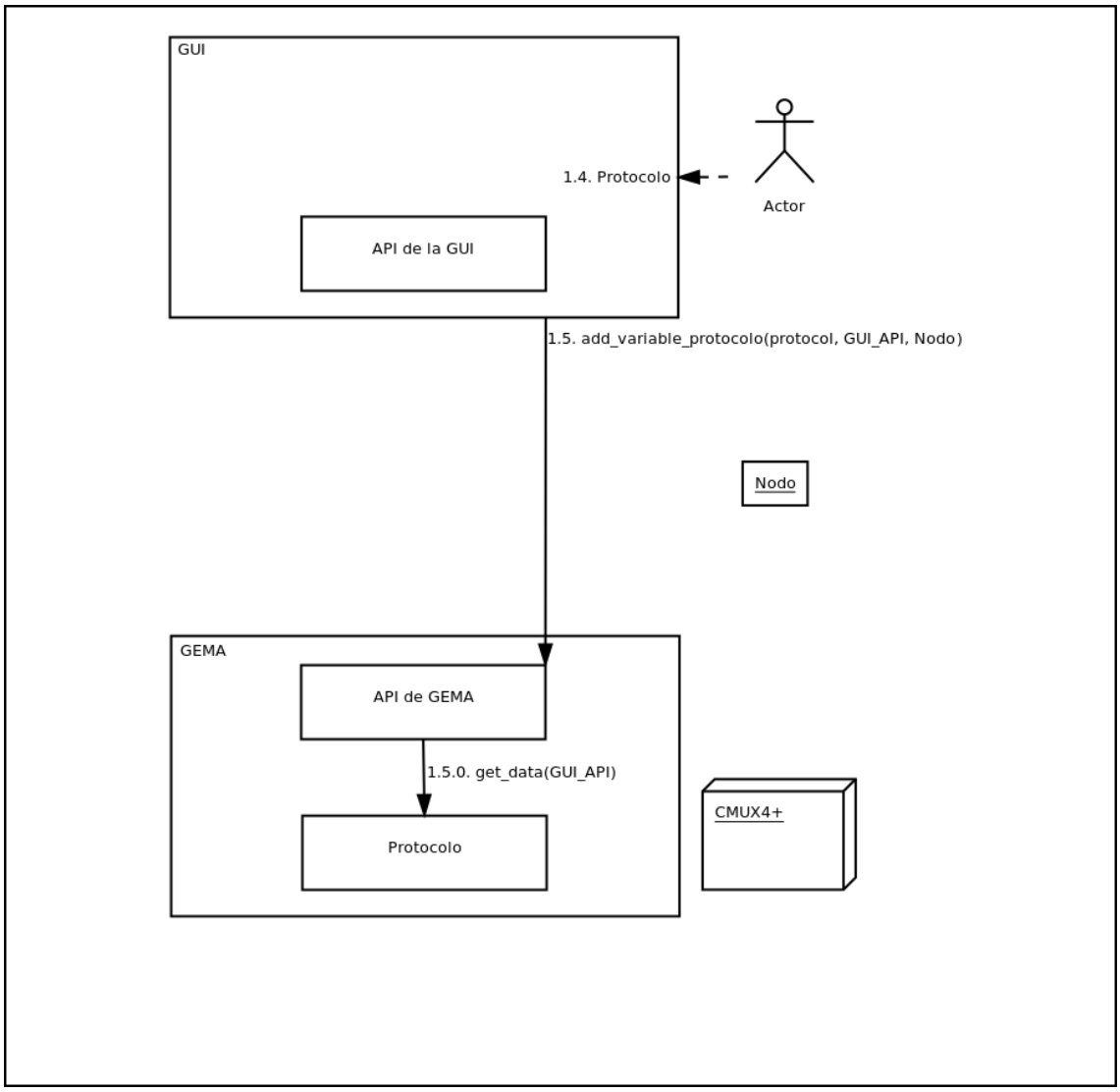


Figura 3.31. Alta de variable SNMPv1 Fase 3.

1.4. El usuario selecciona un protocolo.

1.5. Tras la selección del protocolo, el interfaz inicia el proceso de alta de variable en GEMA indicando junto con el protocolo seleccionado el nodo en el que se deberá dar de alta la variable y una referencia al API de la GUI que será utilizado para la comunicación entre el protocolo y el usuario.

1.5.0. El API de GEMA traslada la petición al protocolo invocando el método ‘get_data’ e indicándole el API de la GUI que deberá utilizar para la comunicación.

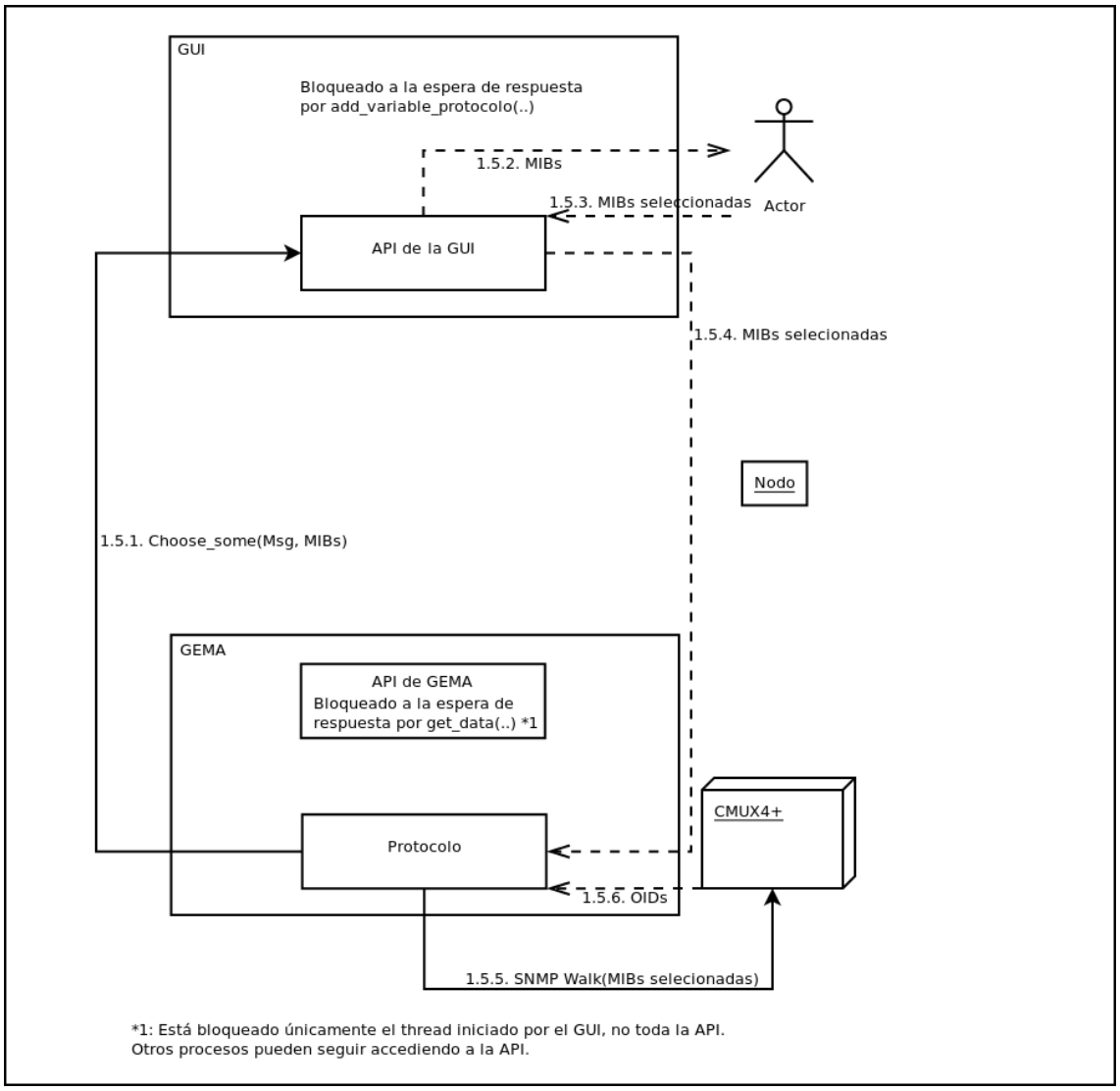


Figura 3.32. Alta de variable SNMPv1 Fase 4.

1.5.1. El protocolo, ejecutando de ‘get_data’ comienza a solicitar información al usuario, solicitando en primer lugar la selección de una serie de MIBs de entre la lista de las disponibles. Las MIBs disponibles están definidas en el fichero de configuración de GEMA.

1.5.2. El interfaz muestra al usuario la lista de MIBs a escoger en el formato que se haya considerado adecuado solicitando una selección múltiple.

1.5.3. El usuario selecciona las MIBs que desea utilizar para monitorizar el dispositivo.

1.5.4. El API de la GUI comunica la selección directamente al protocolo.

1.5.5. El protocolo, ejecutando los métodos internos ‘__checkGet’ y ‘__pysnmp_mib’, recorrerá el dispositivo averiguando las variables disponibles descritas por las MIBs seleccionadas.

1.5.6. El dispositivo comunica al protocolo los OID de todas las variables disponibles.

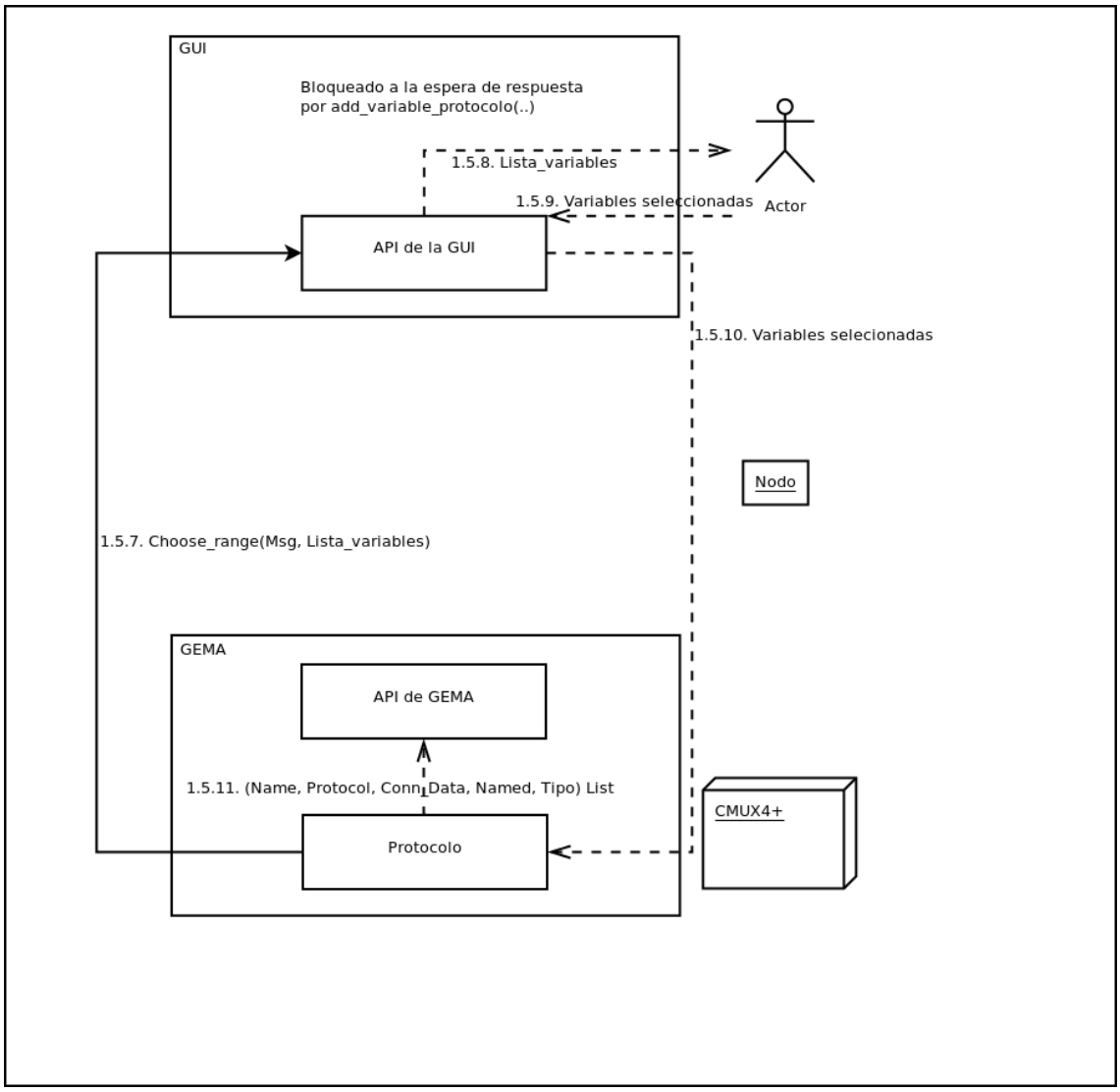


Figura 3.33. Alta de variable SNMPv1 Fase 5.

1.5.7. Una vez obtenida la lista de variables disponibles en el dispositivo y sus OID descriptivos solicita al API de la GUI una selección múltiple de las mismas.

1.5.8. El interfaz muestra la lista al usuario solicitando la selección correspondiente.

1.5.9 El usuario indica las variables que desea añadir al nodo de entre las ofrecidas.

1.5.10. La selección realizada es comunicada directamente al protocolo.

1.5.11. El protocolo construye los Conn_Data de todas las variables seleccionadas y devuelve la lista de datos requeridos para el alta de las variables al API de GEMA.

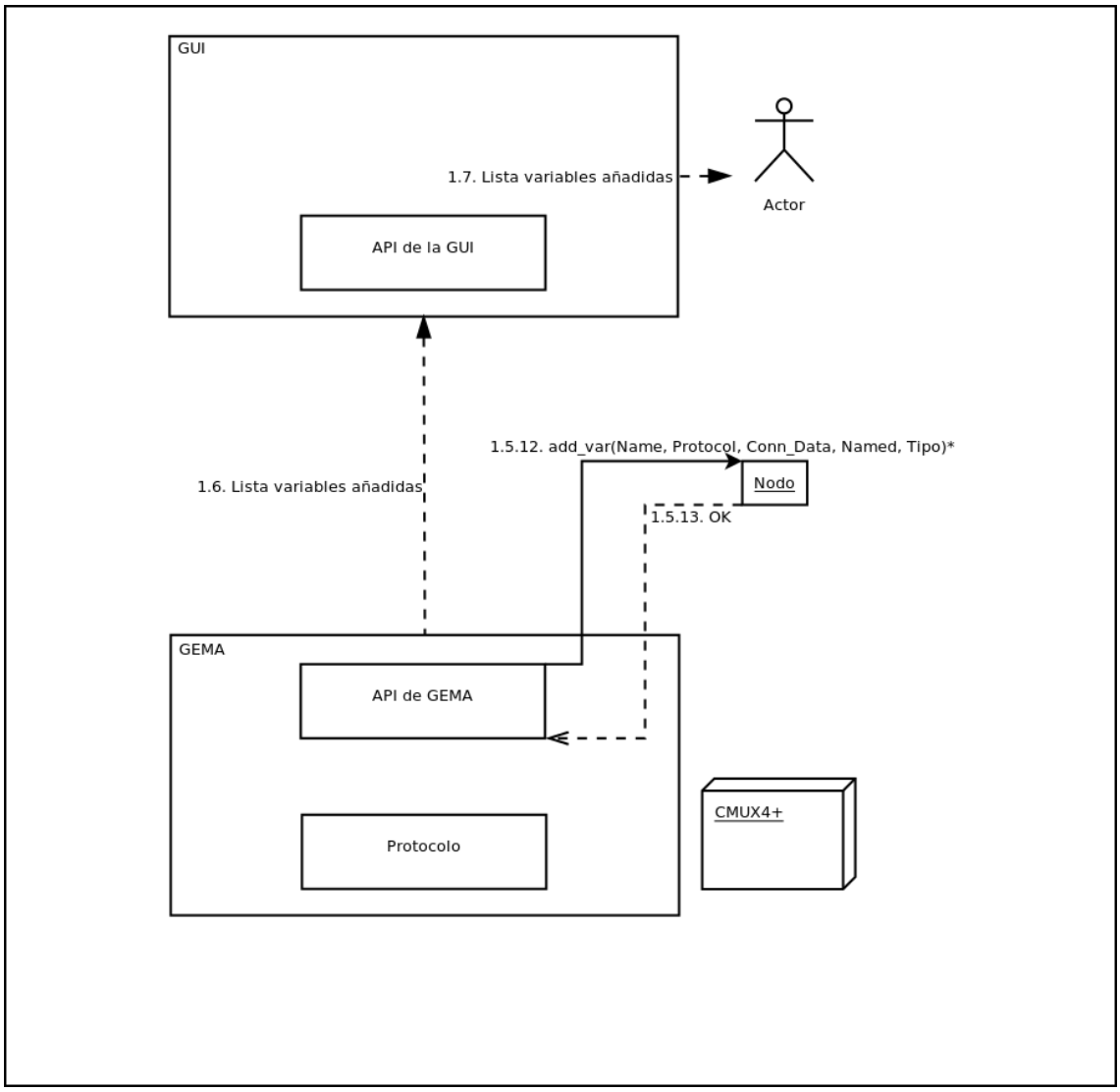


Figura 3.34. Alta de variable SNMPv1 Fase 6.

1.5.12. El API de GEMA, una vez terminado todo el proceso requerido por el protocolo, procede a dar de alta las variables obtenidas en el nodo seleccionado.

1.5.13. El nodo comunica al API la correcta adición de la variable o el error correspondiente en caso de haberlo.

1.6. El API de GEMA devuelve la lista de resultados al interfaz, dando así por finalizada la operación de alta de variables.

1.7. El interfaz comunica al usuario el resultado de la operación.

Tras este proceso todas las variables seleccionadas por el usuario de entre las disponibles en el dispositivo han sido añadidas al sistema. Para ello no ha sido necesario que ningún componente tenga conocimiento de las funcionalidades o comportamientos de cualquier otro. De esta forma cualquier elemento de los involucrados en el proceso es sustituible completamente siempre y cuando el nuevo atienda a un mínimo de llamadas definidas en su interfaz.

La implementación de SNMPv2c, incluida también de serie en GEMA, es en un 99% idéntica a la expuesta (presenta cambios únicamente en 9 líneas de código y el diseño seguido es exactamente el mismo).Presentando únicamente diferencias en los parámetros de llamada a las funciones de librería de PySNMP relativas al cambio de versión. Es por ello que no se incluye una descripción detallada de la misma y se considerará válido para SNMPv2c en todos los casos el diseño expuesto para SNMPv1.

3.5 Diseño e implementación del nivel de presentación

En este nivel se localizan todos los elementos externos conectados a GEMA junto con los mecanismos de comunicación o conexión definidos para ello. Para su comunicación con el núcleo de la aplicación se implementa un API que incluye todas las llamadas necesarias tanto para la administración, configuración y gestión de la red como para las labores de monitorización.

De esta forma se evita que cualquier componente pueda acceder directamente a la estructura interna de GEMA, evitando así tanto la necesidad de que los componentes externos conozcan el diseño interno como de posibles accesos no deseados a su estructura o implementación.

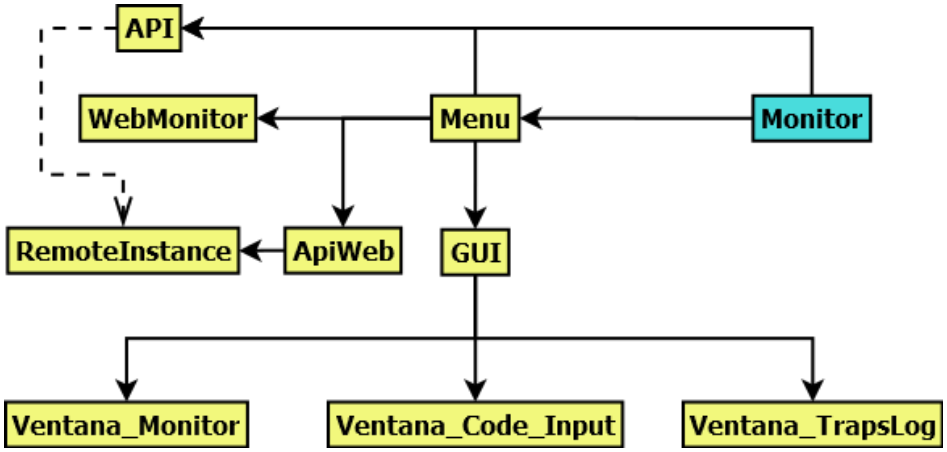


Figura 3.35. Modelo simplificado de clases de la capa de presentación.

Se implementa un API con acceso web que extiende las funcionalidades del API principal permitiendo la solicitud remota de operaciones sobre GEMA. Mediante este API-WEB se implementan aplicaciones de monitorización, gestión y administración remotas.

Cuenta también con un menú que permite administrar todas las funcionalidades desde el equipo local. Este menú permite controlar los diferentes componentes conectados al sistema, así como diseñar, gestionar y monitorizar la red de forma local.

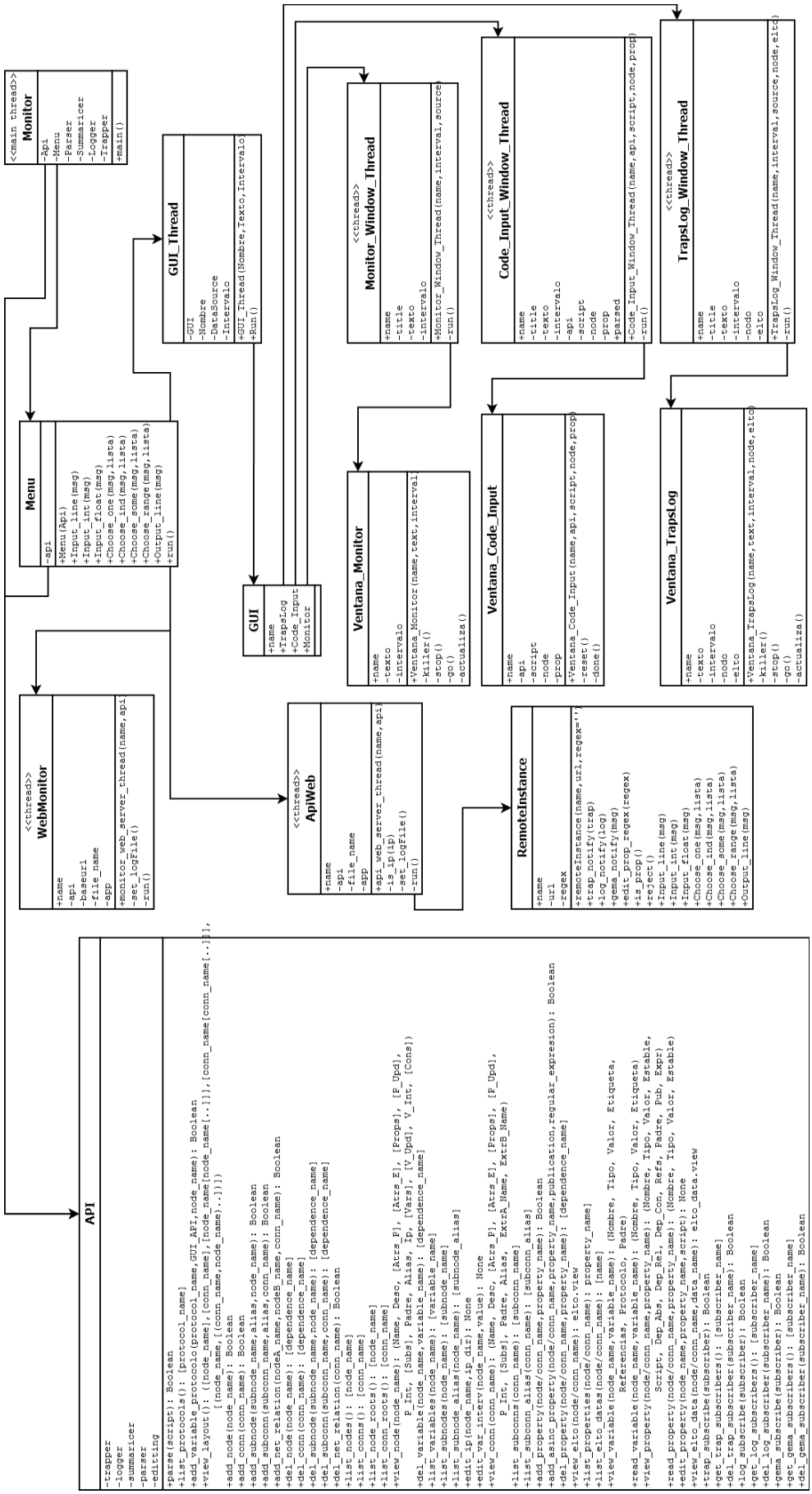


Figura 3.36. Diagrama de clases del nivel de presentación reducido.

Cada elemento activo en este nivel creará su propio hilo de ejecución independientemente del hilo principal de la aplicación de esta forma no interfiere directamente en el funcionamiento de GEMA. Al estar ejecutados en su propio hilo cualquier fallo interno que ocurriese dentro del elemento únicamente afectaría al mismo. Se evita de esta forma que fallos internos en la implementación o ejecución de módulos añadidos afecten al funcionamiento del núcleo.

3.5.1 API

El API es la clase encargada de establecer un mecanismo estándar de comunicación con el exterior. Actúa como frontal de todo el sistema, de forma que toda comunicación con los módulos y aplicaciones externas se realiza a través de él. Esto permite establecer un punto de control a todas las operaciones realizadas, así como implementar cualquier sistema de gestión de permisos, usuarios, grupos, etc.

API
-trapper -logger -summaricer -parser -editting
+apig(parser,summaricer): API -set_logger(logger) -set_trapper(trapper) -legal_name(name) -__transform_refs([name]) +parse(script): Boolean +list_protocols(): [protocol_name] +add_variable_protocolo(protocol_name,GUI_API,node_name): Boolean +view_layout(): ([node_name],[conn_name],[node_name[node_name[..]]],[conn_name[conn_name[..] [(node_name,[(conn_name,node_name)..])]) +read_layout(): view +start_all(): None +stop_all(): None +add_node(node_name): Boolean +add_conn(conn_name): Boolean +add_subnode(subnode_name,alias,node_name): Boolean +add_subconn(subconn_name,alias,conn_name): Boolean +add_net_relation(nodeA_name,nodeB_name,conn_name): Boolean +del_node(node_name): [dependence_name] +del_conn(conn_name): [dependence_name] +del_subnode(subnode_name,node_name): [dependence_name] +del_subconn(subconn_name,conn_name): [dependence_name] +del_net_relation(conn_name): Boolean +list_nodes(): [node_name] +list_conns(): [conn_name] +list_node_roots(): [node_name] +list_conn_roots(): [conn_name] +view_node(node_name): (Name, Desc, [Atrs_P], [Atrs_E], [Props], [P_Upd], P_Int, [Subs], Padre, Alias, Ip, [Vars], [V_Upd], V_Int, [Cons]) +read_node(node_name): (Name, Alias, ExtrA_Name, Ip) +start_node(node_name): None +stop_node(node_name): None +del_variable(node_name,variable_name): [dependence_name] +list_variables(node_name): [variable_name] +list_subnodes(node_name): [subnode_name] +list_subnode_alias(node_name): [subnode_alias] +edit_ip(node_name,ip_dir): None +edit_var_interv(node_name,value): None +edit_URL(node_name,URL): None +edit_GIS(node_name,GIS): None +edit_IdCh(node_name,IdCh): None +dismiss_ip(node_name): None +node_ip(node_name): Ip +view_conn(conn_name): (Name, Desc, [Atrs_P], [Atrs_E], [Props], [P_Upd], P_Int, [Subs], Padre, Alias, ExtrA_Name, ExtrB_Name)

```
+read_conn(conn_name): (Name, Alias, ExtrA_Name, ExtrB_Name)
+start_conn(conn_name): None
+stop_conn(conn_name): None
+edit_long(conn_name,longitud): None
+edit_sent(conn_name,sentido): None
+edit_disp(conn_name,disponibilidad): None
+list_subconns(conn_name): [subconn_name]
+list_subconn_alias(conn_name): [subconn_alias]
+add_property(node/conn_name,property_name): Boolean
+add_asinc_property(node/conn_name,property_name,publication,regular_expresion): Boolean
+del_property(node/conn_name,property_name): [dependence_name]
+view_elto(node/conn_name): elto.view
+read_elto(node/conn_name): elto.read
+edit_desc(node/conn_name,descripción): None
+edit_interv(node/conn_name,value): None
+list_properties(node/conn_name): property_name]
+list_elto_datas(node/conn_name): [name]
+start_elto_data(node/conn_name,var/prop_name): Boolean
+stop_elto_data(node/conn_name,var/prop_name): Boolean
+view_variable(node_name,variable_name): (Nombre, Tipo, Valor, Etiqueta,
                Referencias, Protocolo, Padre)
+read_variable(node_name,variable_name): (Nombre, Tipo, Valor, Etiqueta)
+view_property(node/conn_name,property_name): (Nombre, Tipo, Valor, Estable,
                Script, Dep_Abs, Dep_Rel, Dep_Con, Refs, Padre, Pub, Expr)
+read_property(node/conn_name,property_name): (Nombre, Tipo, Valor, Estable)
+edit_property(node_name,property_name,script): None
+view_elto_data(node/conn_name,data_name): elto_data.view
+read_elto_data(node/conn_name,data_name): elto_data.read
+dump_layout(file_name): None
+load_layout(file_name): None
+reset_layout(): None
+trap_subscribe(subscriber): Boolean
-trap_notify(trap)
+trap_file_purge(ip): none
+get_trap_subscribers(): [subscriber_name]
+del_trap_subscriber(subscriber_name): Boolean
+start_trapper(): None
+stop_trapper(): None
+log_subscribe(subscriber): Boolean
-log_notify(log)
+log_file_purge(ip): None
+get_log_subscribers(): [subscriber_name]
+del_log_subscriber(subscriber_name): Boolean
+start_logger(): None
+stop_logger(): None
+gema_subscribe(subscriber): Boolean
-gema_notify(msg)
+get_gema_subscribers(): [subscriber_name]
+del_gema_subscriber(subscriber_name): Boolean
+edit_regex(subscriber_name,new_regular_expresion): None
-is_ip(ip)
+list_methods(): [method_name]
```

Figura 3.37. Clase API.

La mayoría de los métodos implementados simplemente propagan las llamadas a los respectivos métodos del Layout. Se implementan una serie de métodos internos para ayudar a asegurar la corrección de los datos de entrada verificando el tipo y formato de los mismos antes de pasarlos al núcleo.

El API incluye referencias y métodos de acceso a los módulos de sistema tales como el parser o el sistema de serialización de ficheros. De esta forma se permite la interactuación con estos módulos desde el exterior de una forma controlada y supervisada.

3.5.2 API-Web

El API-Web extiende los servicios y funciones proporcionados por el API de forma remota. Establece un servicio web en el cual, mediante llamadas POST y GET una aplicación externa puede comunicarse con GEMA. Todas las peticiones recibidas, una vez procesadas y verificadas, son transmitidas al API. El resultado devuelto por el API es comunicado como respuesta a la solicitud realizada por la aplicación remota.

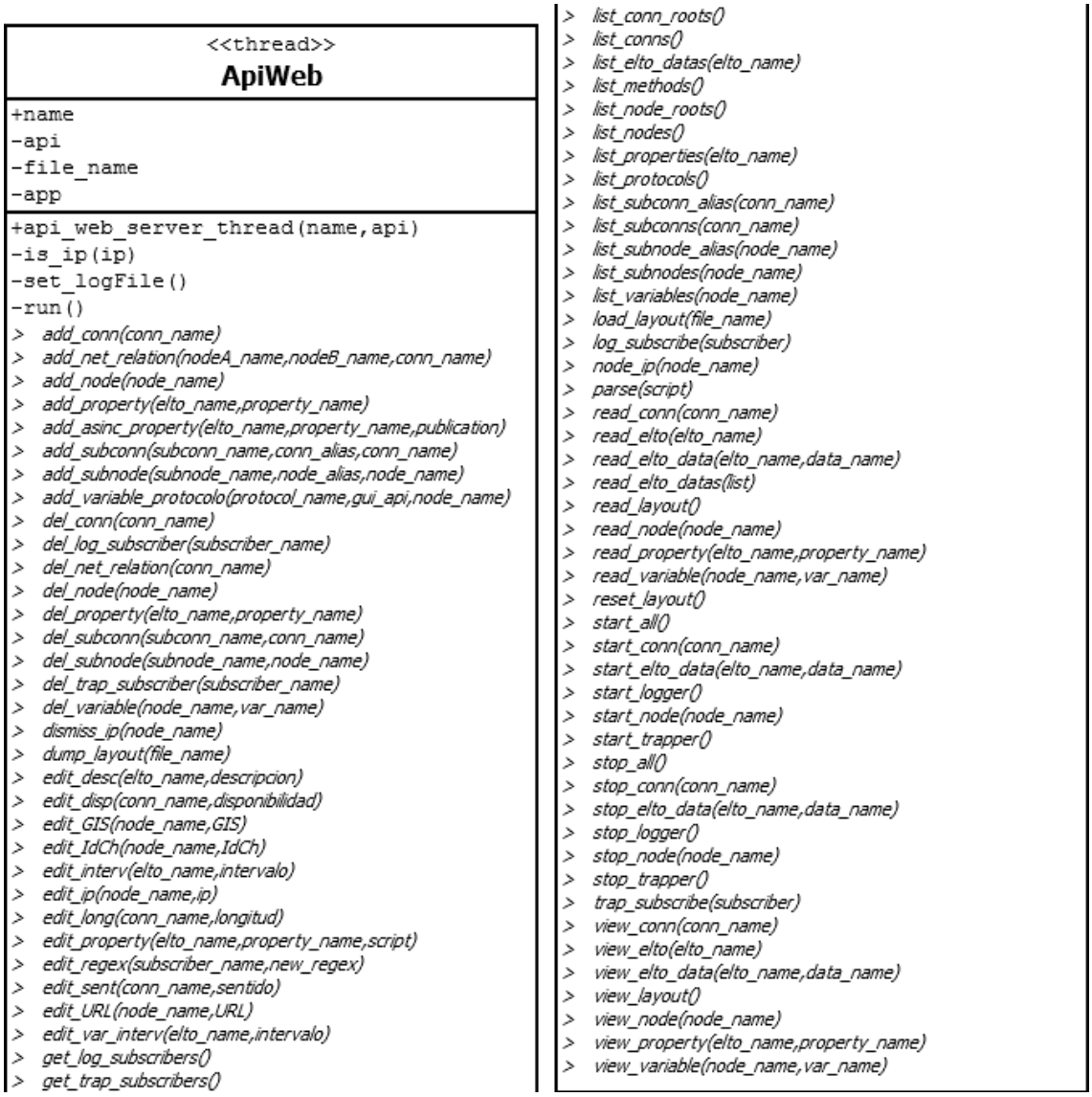


Figura 3.38. Clase API-Web.

De esta forma el API-Web resulta en un mero intermediario que traduce peticiones de un entorno a otro. Con esto se consigue independizar la implementación interna de GEMA, incluso de su API, del desarrollo de aplicaciones de monitorización remotas apoyadas en la plataforma de GEMA.

El servidor del API-Web se ejecutará en su propio thread, de esta forma su funcionamiento se independiza completamente del hilo de ejecución de la aplicación.

3.5.3 Representante Remoto (remote instance)

Los representantes remotos son una representación local de un subscriptor o API-Gui de un elemento externo conectado a la aplicación con el cual GEMA deberá dialogar. Cuando el API-Web recibe una petición a estos servicios (subscripción o alta de variables) crea, con los datos proporcionados, una instancia de esta clase.

Esta clase es pasada como parámetro al API de forma que desde el punto de vista del API todos los elementos conectados son locales al sistema. De esta forma se consigue independizar el funcionamiento de los métodos del API del medio de conexión real con el elemento externo.

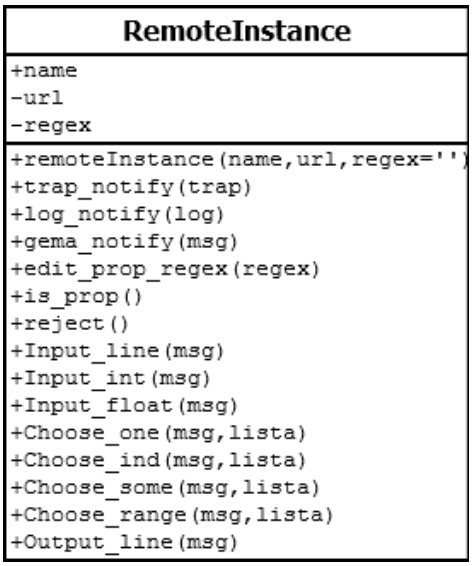


Figura 3.39. Clase RemoteInstance.

Esta clase implementa todos los métodos que se deben encontrar en los elementos conectados. Internamente la clase realizará directamente las peticiones al subscriptor o API-GUI representados y devolverá las respuestas proporcionadas. De esta forma, una vez dados de alta la comunicación es independiente de cualquier otro módulo de GEMA.

Para su correcto funcionamiento los elementos externos deberán haber proporcionado una dirección web, “url”, en la que atender las peticiones POST que se realizarán.

Los objetos de esta clase serán los proporcionados tanto al Dispatcher de notificaciones como a los protocolos durante el proceso de alta de variables. Actuando a todos los efectos como un elemento local dentro del sistema de la aplicación.

Constan también de una expresión regular, “regex”, que es utilizada por el Dispatcher para determinar que notificaciones le corresponden. Esta expresión es modificable, de forma que es posible cambiar el conjunto de notificaciones a los que está subscrito sin necesidad de volver a dar de alta la subscripción.

3.5.4 Menú

El menú es el punto inicial de comunicación con el usuario. Es un interfaz de consola local iniciado directamente con la ejecución de la aplicación. Desde el, además de todas las opciones de uso de GEMA disponibles se controla la activación de los diferentes componentes. Es el encargado de activar, bajo demanda del usuario, el API-Web y el servidor web de monitorización local.

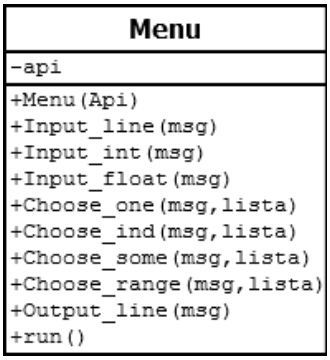


Figura 3.40. Clase Menú.

El menú se conecta con GEMA haciendo uso del API de forma que todas las peticiones al núcleo siguen encauzadas a través de esta. Implementa así mismo los métodos requeridos para la API-GUI durante el proceso de alta de variables. Desde el punto de vista interno de GEMA es un Interfaz con el usuario más y es tratado de igual forma.

El menú es un interfaz orientado a consola. Una vez lanzada la ejecución de GEMA se mostrará en pantalla una serie de menús en formato texto. Esto permite la ejecución de GEMA desde un simple terminal, e incluso, su administración desde un terminal remoto como SSH.

Desde este menú se podrá lanzar el interfaz gráfico local. El menú incorpora un pequeño interfaz gráfico de monitorización. Este interfaz incorpora una serie de ventanas que permiten la visualización de la red, editar el código de las propiedades y monitorizar los Traps y Logs recibidos.

Este interfaz hereda del primer prototipo de visualización implementado durante los ciclos de evolución del núcleo. Inicialmente conectaba directamente con el Layout pero una vez diseñada e implementada el API se modificó su código para que trabajase a través de ella. Es por ello un ejemplo de uso e implementación de módulos añadidos al sistema a través del API.

Todo el interfaz ejecuta desde su propio hilo, independizándolo así del menú y permitiendo su trabajo en paralelo. El objetivo de implementar todos los elementos lanzados desde el menú en hilos independientes es garantizar la continuidad del flujo de trabajo del núcleo de GEMA y la disponibilidad del menú de consola para su acceso. Asegurando estos elementos podemos garantizar una monitorización continua y la ejecución de las propiedades programadas en el sistema independientemente de cualquier fallo que pueda producirse en el resto de módulos.

La persistencia del menú en la consola de ejecución permitirá un acceso directo a GEMA con independencia de cualquier modulo o interfaz añadidos.

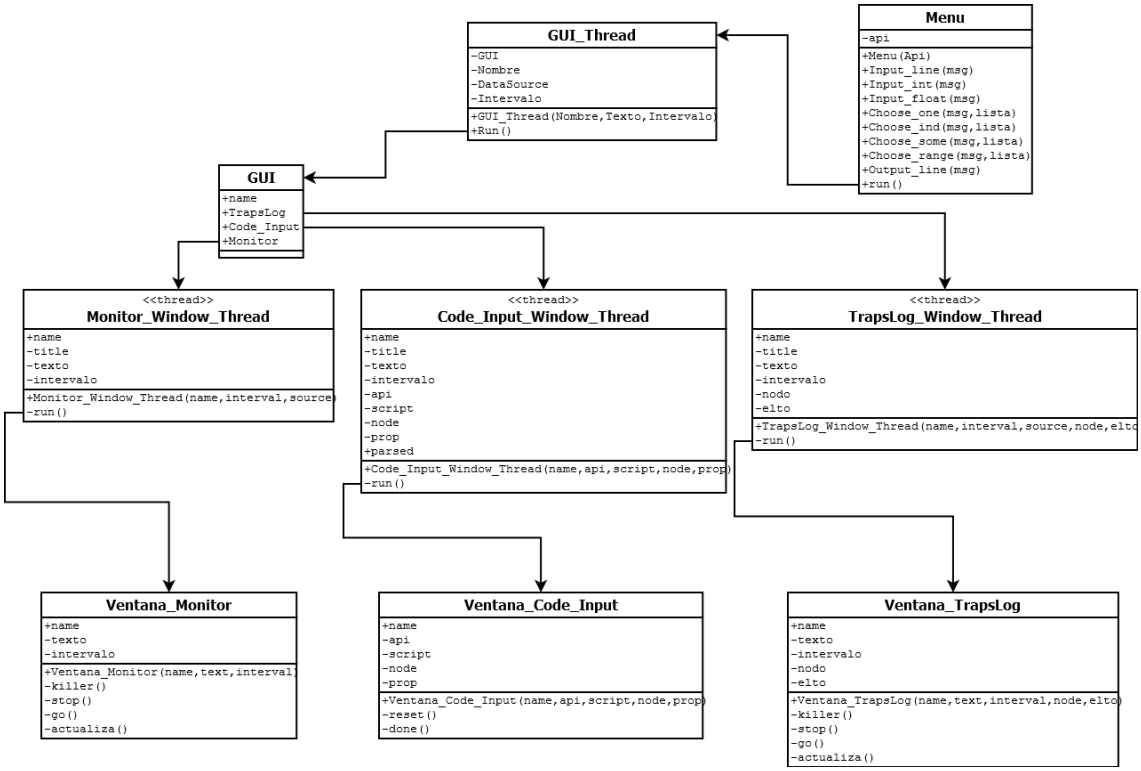


Figura 3.41. Diagrama de clases del GUI del menú.

Cada una de estas ventanas incorpora su propio thread de actualización de forma que su funcionamiento, una vez inicializadas, es independiente al menú.

Para la implementación de este interfaz se han utilizado las librerías Tcl/Tk que incorpora Python [28]. El objetivo de este interfaz no es el uso completo de la aplicación si no un recurso adicional de control desde el que verificar el correcto funcionamiento. También permite visualizar una información concreta para el diagnostico de posibles fallos o errores en el sistema.

Finalmente, el menú incluye una consola interactiva de Python con acceso al estado del intérprete. Esta consola permitirá leer y modificar cualquier elemento de GEMA de forma directa e, incluso, modificar su comportamiento en tiempo de ejecución. El acceso se realiza al más bajo nivel, ejecutando directamente sentencias Python contra las estructuras de datos directamente en memoria.

El uso de esta consola está destinado a tareas de depuración y mantenimiento pero puede servir de acceso directo, o puerta trasera, para tratar casos críticos que requieran una evaluación y solución en caliente. Sin embargo, para poder aprovechar su potencial exige un amplio conocimiento del diseño y la implementación interna de GEMA así como del lenguaje de programación Python.

3.5.5 Monitor Web

El monitor web implementa un servidor web en la propia máquina. Este servicio permitirá a dispositivos remotos acceder a servicios de monitorización directamente desde un navegador. Así se permite el acceso remoto a GEMA sin necesidad de distribuir e instalar aplicaciones adicionales.

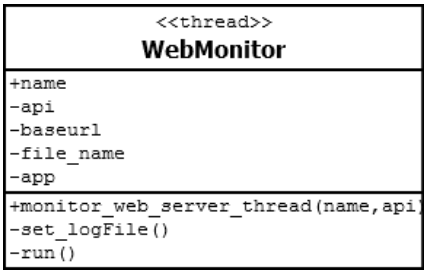


Figura 3.42. Clase Monitor Web.

El monitor, como el resto de los complementos, ejecutará en su propio hilo de ejecución. El monitor web es uno de los servicios que deben activarse desde el menú de consola antes de estar disponible para su uso.

Para su implementación se ha utilizado el microframework Flask [29] de Python. Este framework generará las páginas HTML [30] que serán publicadas por el servidor web.

Toda la información de configuración necesaria para su funcionamiento: host, url, puerto, etc. estará definida en el módulo de configuración "Config".

3.5.6 Aplicación remota

Las aplicaciones remotas son cualquier software que se comunica con GEMA a través del API-Web. Para su implementación únicamente es necesario conocer el API-Web ofrecido. Toda la información proporcionada desde GEMA por este interfaz está en un formato crudo y serializado. Estas aplicaciones deben encargarse de su interpretación y todos los aspectos de visualización y representación de la información al usuario.

Dada la completa independencia de estas aplicaciones con respecto a GEMA no se hace ningún modelo o diseño de las mismas, limitándose únicamente a la definición del API-Web. Para todas las operaciones que requieran una interacción bidireccional la aplicación será representada en GEMA mediante una "remote Instance" como se vio anteriormente.

El resto de operaciones realizadas por la aplicación remota son o consultas que recibirán una respuesta inmediata a la petición solicitada u operaciones de modificación del sistema que serán procesadas inmediatamente, devolviendo su resultado como respuesta a la petición.

Ejemplos de estas aplicaciones pueden ser clientes en cualquier máquina remota que ofrecen las funcionalidades completas de GEMA, clientes en dispositivos móviles para seguimiento de la monitorización o recepción de notificaciones, incluso un posible gestor de nivel superior que intercomunique dos o más instancias de GEMA entre ellas u otros servicios o sistemas de gestión y monitorización de redes.

3.6 Diseño e implementación de la capa de sistema

La capa de sistema contiene todos los módulos internos que interactúan directamente con GEMA y con el sistema en el que se ejecuta. Estos módulos, considerados internos a GEMA, se implementan de forma independiente al núcleo de la aplicación. Son los responsables de proporcionar funcionalidad a GEMA más allá de la mera monitorización y actuación sobre la red a gestionar.

La capa de sistema, a diferencia de las demás presentadas, no consta de un modelo conexo de clase, si no que agrupa diferentes estructuras de clases que describen a los diferentes servicios añadidos al núcleo de la aplicación.

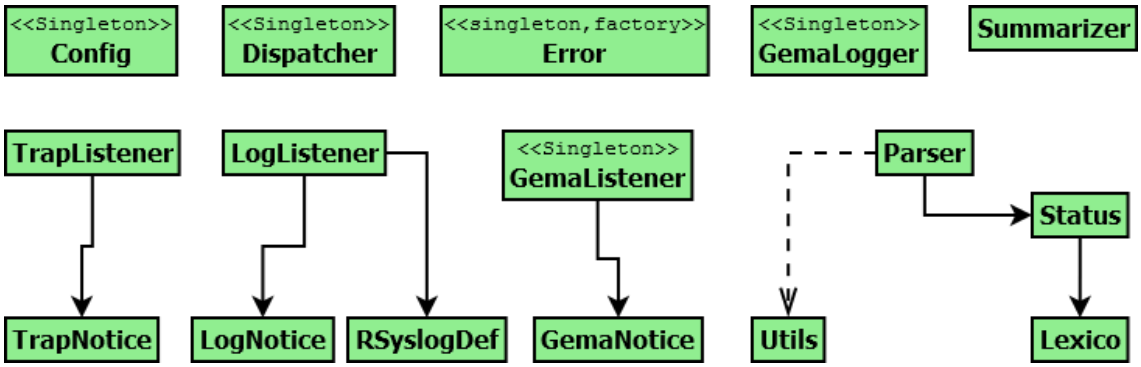


Figura 3.43. Modelo simplificado de clases de la capa de sistema.

Podemos observar una serie de módulos generales definidos bajo una clase con un patrón singleton. Estos módulos realizan operaciones internas de GEMA y no son controlados por los usuarios como el Dispatcher, encargado de distribuir las notificaciones generadas o Gema Logger, encargado de generar los logs de la aplicación.

El patrón singleton en estos módulos se debe a que su presencia en la aplicación se da por supuesta y pueden ser accedidos desde cualquier clase que precise de sus funcionalidades.

Aquellos módulos que no dependen de una clase singleton estarán referenciados a través del API. Estos módulos ofrecen servicios a los usuarios de la aplicación tales como el salvado de layouts mediante el Summarizer o el compilado de códigos GEMA-Script mediante el Parser.

Para la implementación y adición de nuevos módulos en este nivel es posible utilizar cualquiera de las dos estrategias según el modelo de servicio que ofrezcan:

Aquellos módulos que ofrezcan un servicio general accesible por cualquier elemento se implementarán como singleton, utilizando el mecanismo descrito anteriormente. Una vez implementados únicamente deberán ser importados y utilizados en aquellos módulos que los requieran dentro de GEMA.

Aquellos módulos que proporcionen funcionalidades accesibles por los usuarios o que deban ser accedidos a través del API se implementarán normalmente. Una vez implementados serán instanciados en el arranque de GEMA y proporcionada su referencia al API.

Para su utilización la API deberá ser modificada adecuadamente tanto para mantener una referencia constante a estos módulos como para ofrecer los métodos de acceso necesarios.

Estos cambios en el API deberán ser propagados al API-Web para permitir su acceso por las aplicaciones externas.

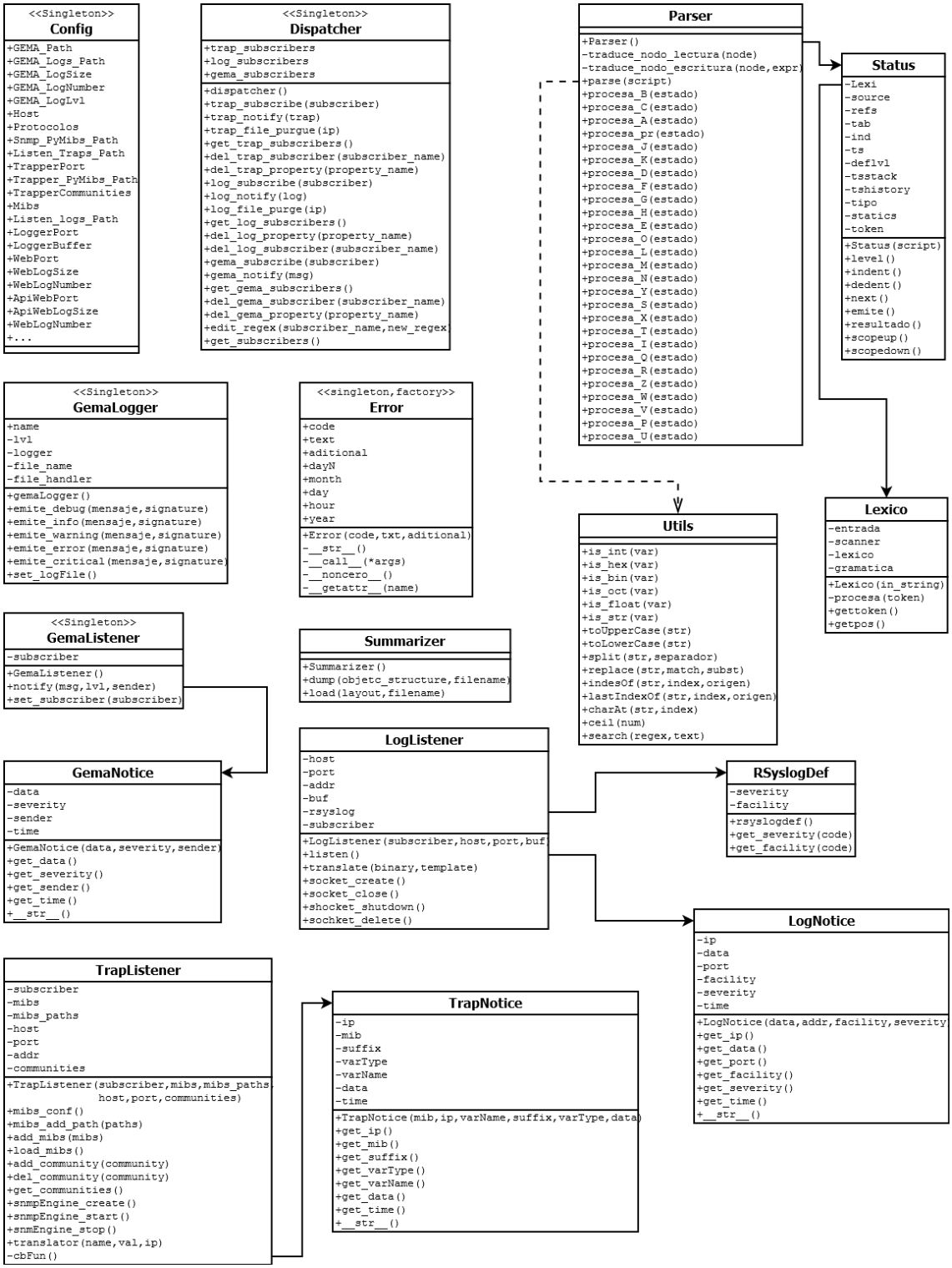


Figura 3.44. Diagrama de clases del nivel de sistema reducido.

3.6.1 Summarizer

El modulo Summarizer es el encargado de la serialización y almacenamiento en disco de los layouts. Su implementación se realiza en base al módulo de marshalling de Python "Pickle".

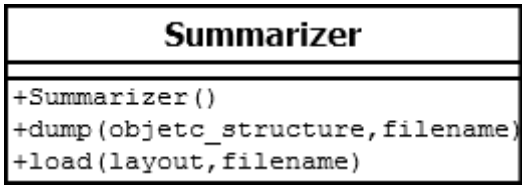


Figura 3.45. Clase Summarizer.

Proporciona dos únicos métodos:

- El método "Dump" recibe una referencia a un objeto, que deberá poseer el método "get_summary". La ejecución de este método en el objeto referenciado deberá devolver una estructura que será serializada mediante la librería Pickle y almacenado en disco. Para esta serialización se dispone de diferentes métodos, desde su codificación en un fichero de texto hasta diferentes mecanismos de codificación binaria con diferentes niveles de optimización.

  La existencia del módulo Summarizer como intermediario entre GEMA y el módulo Pickle permite aplicar a la serialización diferentes métodos de cifrado, seguridad, gestión de versiones y control de errores de una forma completamente independiente tanto a GEMA como a Pickle.

- El método "Load" cargará el objeto serializado almacenado en el fichero dentro de un objeto recibido como parámetro (layout). Este objeto deberá constar con el método "set_summary" que deberá ser capaz de, partiendo del objeto serializado, reconstruir en memoria toda la estructura del objeto original serializado.

  Si el objeto a serializar contuvieses una estructura de objetos anidada o jerarquía de los mismos, como ocurre en el caso de los layouts, debe ser este mismo el que resuelva estas dependencias dentro de sus métodos get_summary y set_summary. Así el módulo Summarizer es completamente independiente de la implementación interna de las estructuras de objetos dentro de GEMA.

3.6.2 Dispatcher

El Dispatcher es el encargado de gestionar todas las subscripciones y distribuir las diferentes notificaciones que se produzcan a sus respectivos destinatarios.

Internamente consta de un diccionario por cada categoría de subscripción disponible: Traps, Logs y Mensajes GEMA. De esta forma los espacios de nombres de las diferentes categorías son independientes permitiendo así que un mismo subscriptor esté presente en varias de ellas.

Para cada una de estas categorías dispone de un conjunto de métodos que permiten el alta, baja, gestión y notificación de mensajes en cada grupo.

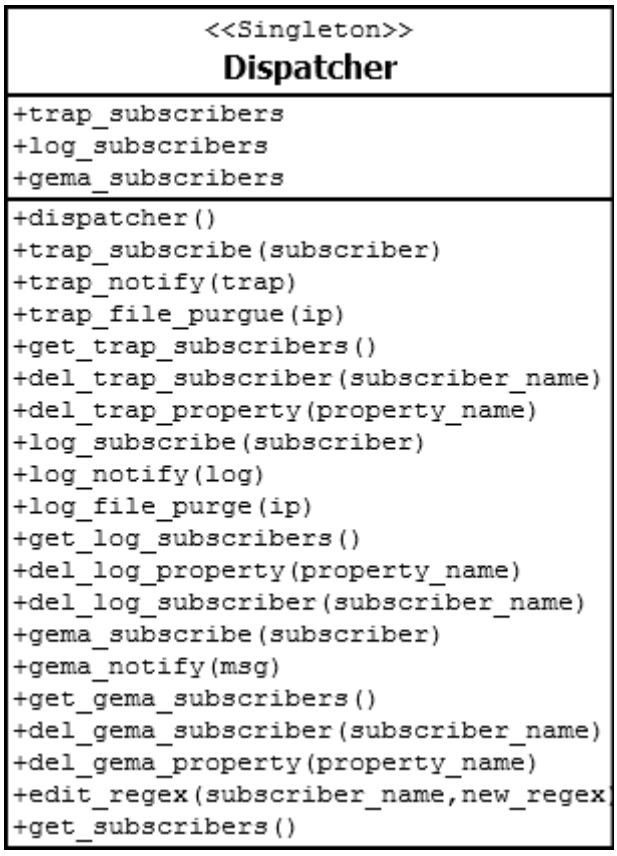


Figura 3.46. Clase Dispatcher.

Para el tratamiento de una notificación se recorrerá la lista de subscriptores cotejando la notificación recibida con la expresión regular de cada uno. Para aquellos subscriptores en que la evaluación de la expresión regular sea correcta se creará un thread exclusivo que se encargará de notificar el mensaje.

De esta forma las notificaciones se realizan de manera asíncrona no afectando así los tiempos de espera de red para subscriptores remotos o tiempos de ejecución de las propiedades asíncronas disparadas.

Posteriormente a la creación del thread de notificación el Dispatcher almacenará en un fichero el mensaje recibido. De esta forma los administradores podrán visualizar y analizar de forma independiente a GEMA todas las notificaciones tratadas. En el caso de los Logs y los Traps estos ficheros estarán identificados por su IP y serán accedidos por los nodos ante una solicitud de su historial de Traps y Logs.

Existe un acoplamiento entre estos módulos en la definición de estos nombres de fichero. Esta decisión de diseño se toma al conceder mayor prioridad a la persistencia de la información y su acceso que al diseño. De este modo un Nodo es capaz de acceder a su historial de Traps y Logs incluso en caso de fallo en el módulo Dispatcher.

Esta clase se implementa según el patrón singleton mediante la estrategia definida anteriormente. El SingleDispatcher proporciona métodos para reiniciar y destruir el dispatcher actual.

3.6.3 TrapListener

El TrapListener controla el servicio de escucha y captura de Traps de GEMA. Ejecuta dentro de su propio thread y se apoya en las librerías PySNMP de Python para su implementación.

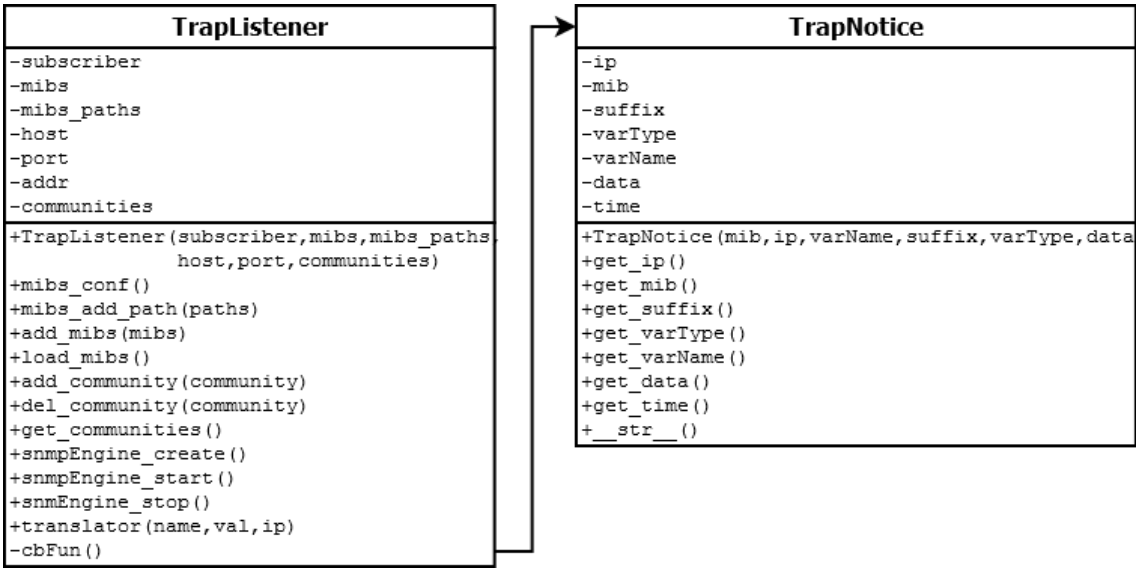


Figura 3.47. Diagrama de clases del TrapListener.

Tras su inicialización creará un servicio de escucha con la configuración proporcionada por el módulo Config y quedará a la espera de la llegada de Traps de la red.

La clase dispone de los métodos necesarios para actualizar la lista de communities permitiendo así la escucha y tratamiento de Traps procedentes de nuevos nodos dados de alta en el sistema. Dispone también de métodos para la carga y adición de MIBs al servicio de escucha de forma que se puedan traducir las notificaciones de nuevos modelos de dispositivos incorporados.

Una vez capturado un Trap la función de "callback" (cbFun) creará una instancia de la clase TrapNotice. Esta instancia será proporcionada al dispatcher para su propagación a los subscriptores correspondientes.

La clase TrapNotice consiste básicamente en una descomposición de los diferentes campos del Trap a la que se añade un "timestamp" de recepción en GEMA. Mediante la implementación del magic method __str__ se consigue que esta clase pueda ser tratada directamente como una cadena de texto, simplificando así su tratamiento por módulos externos que no requieran una descomposición de la información contenida.

3.6.4 LogListener

El LogListener es el encargado de mantener el servicio de escucha y captura de Logs de la red. Al igual que el TrapListener ejecuta dentro de su propio thread. Para su implementación se utiliza directamente la clase 'socket' de Python. Importa al módulo RSysLogDef encargado de traducir los 'facility' y 'severity' de los Logs recibidos.

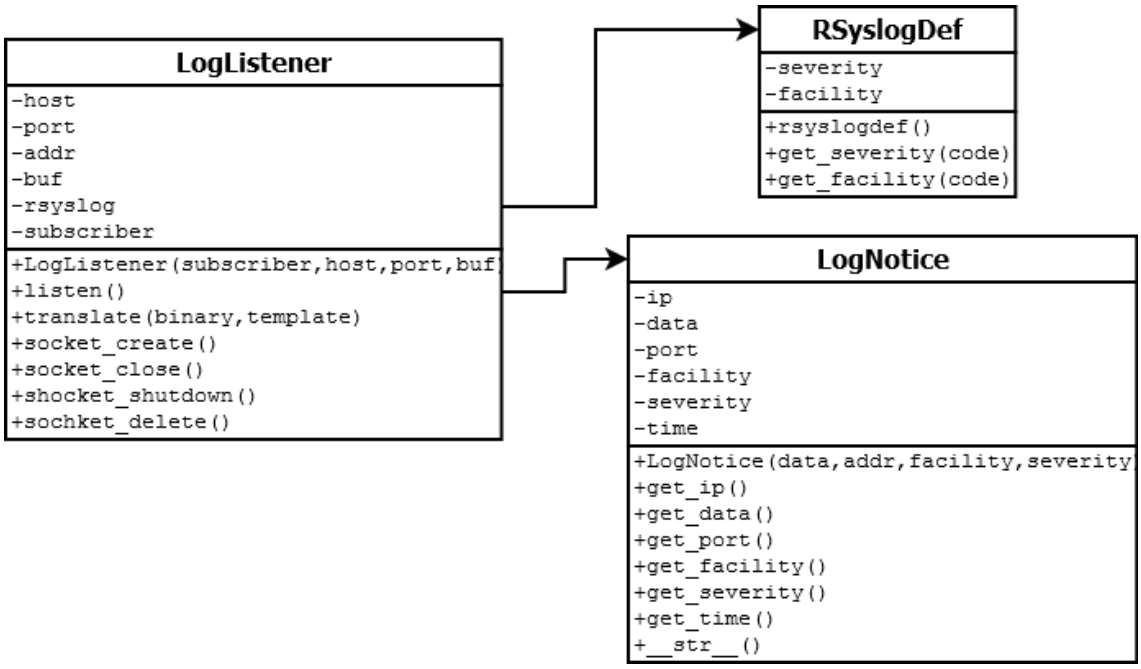


Figura 3.48. Diagrama de clases del LogListener.

Su funcionamiento es similar al TrapListener. Una vez iniciado queda a la escucha de Logs en el puerto definido por el módulo de configuración. Tras recibir un Log crea una instancia de la clase LogNotice con ayuda del RSysLogDef. Esta instancia de LogNotice es pasada al Dispatcher para su distribución a los subscriptores.

La clase LogNotice, al igual que en el caso anterior, consiste en una descomposición de los diferentes campos junto con un "timestamp" de recepción en GEMA. Se realiza una implementación similar del magic method __str__ para poder ser tratada directamente como una cadena de texto.

3.6.5 GemaListener

El modulo GemaListener se encarga de recibir las solicitudes de creación de mensajes GEMA. A diferencia de los dos módulos anteriores estas notificaciones no son capturadas del exterior si no que se generan directamente por la aplicación. Por ello se implementa mediante un patrón singleton que permite a cualquier elemento de GEMA crear una nueva notificación sin la intervención de ninguna otra clase.

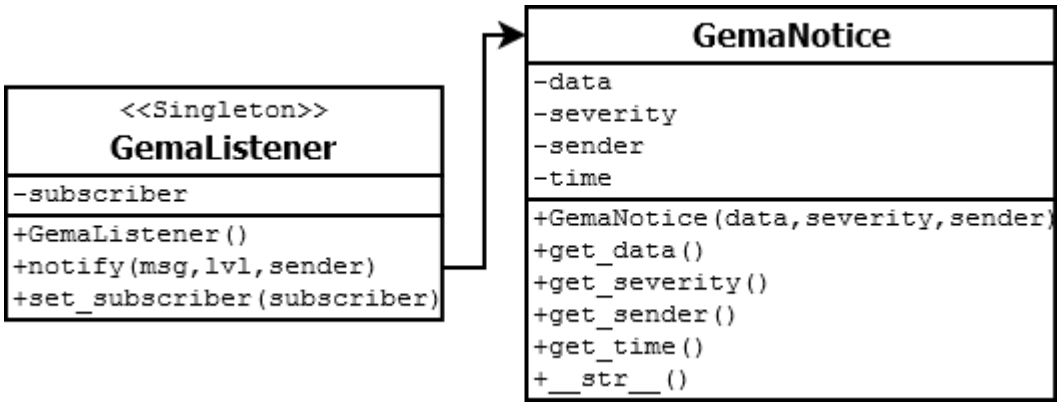


Figura 3.49. Diagrama de clases del GemaListener.

Su implementación interna es sencilla, dispone de un único método para solicitar la creación del mensaje y su notificación. Este método, "notify", creará una instancia de la clase GemaNotice que será pasada al Dispatcher para su tratamiento y distribución.

La clase GemaNotice consiste en una serie de campos que contiene la información del mensaje junto con su "timestamp". Implementa el magic method __str__ de la misma forma que los anteriores para su tratamiento simplificado como cadena de texto.

3.6.6 Error

La clase Error es utilizada para la notificación y propagación de los diferentes errores de GEMA. Estos pueden representar errores en los parámetros de un método, fallos en el uso de servicios e incluso excepciones ocurridas en la ejecución de GEMA.

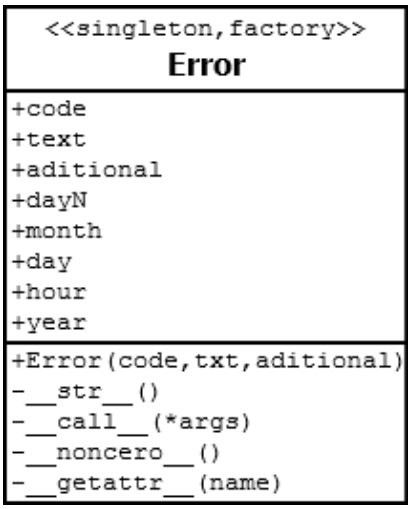


Figura 3.50. Clase Error.

Cada error será representado por una instancia de esta clase y, normalmente, devuelto como respuesta al método que lo ha generado. Internamente contiene un código de error definido, un mensaje textual del error y la información adicional necesaria para su tratamiento. Incluyen también un "timestamp" descompuesto para poder cotejarlo con los diferentes logs generados.

Internamente su implementación se apoya en magic methods para permitir su tratamiento incluso por operaciones que no esperaban recibir un error:

__str__, como ya se ha expuesto anteriormente, permite su tratamiento simple como cadena de texto.

__noncero__ permite su evaluación booleana, devolviendo 'False' en todos los casos.

__call__ permite la ejecución del error como si se tratara de un método o función con cualesquiera parámetros proporcionados. Se devolverá a sí mismo en todos los casos.

__getattr__ permite un intento de acceso a cualquier atributo o método no definido en la clase. Nuevamente se devolverá a sí mismo en todos los casos.

Estos magic methods permiten que el error sea tratado como si se tratara de la respuesta esperada por el elemento que invocó el método y sea propagado por el sistema hasta ser capturado.

De no ser capturado alcanzaría el nivel superior de anidación, el intérprete de Python y sería impreso por consola. Evitando así que provoque la parada de cualquier thread o de la aplicación al completo.

Este comportamiento difiere del de las excepciones. Las excepciones no se propagan entre threads, escalan la jerarquía de llamadas hasta ser tratadas o alcanzar la cima de su hilo de ejecución. Una vez la excepción alcanza la cima esta es impresa en consola junto con un extracto de la pila de llamadas y provoca la parada del hilo de ejecución.

La clase Error implementada evita este comportamiento, no deteniendo en ningún caso la ejecución de un thread y permitiendo pasar de un objeto a otro como si se tratase de una respuesta normal a un método solicitado. Evitando así no solo la detención de servicios de GEMA si no que permite propagar estos errores a través del API hasta las aplicaciones externas o remotas gracias al tratamiento en modo texto proporcionado por le magic method __str__.

Existe una definición estandarizada para los códigos de error de forma que puedan ser identificados directamente por cualquier elemento que los capture.

El código de error consiste en una serie de 6 números “00.00.00.00.00.00”. Los tres primeros valores indican que elemento es responsable del error:

- El primer número indica el módulo en el que es definido y generado el error.
- El segundo número indica la clase dentro del módulo que ha generado el error.
- El tercero indica el método que ha generado el error.

Los dos siguientes números indican el tipo y categoría del error respectivamente, por ejemplo:

Error de permisos de sistema	*09*
Fichero no accesible	*01*

El último número es un simplemente un ordinal, un código interno al método que indica de que error se trata.

De este modo el código de error ‘03.01.11.09.01.01’ indicaría lo siguiente:

03 Módulo Node.
01 Clase Node.
11 Método “get_componente”.
09 Error de permisos de sistema.
01 Fichero no accesible.
01 1er error, en lectura del fichero de Traps.

La definición completa de los códigos de error y sus correspondencias está incluida en la documentación de la aplicación.

3.6.7 GemaLogger

GemaLogger es el módulo encargado de generar las entradas del log de ejecución de GEMA. Esté módulo es accesible desde cualquier elemento al ser implementado como singleton. Cualquier elemento podrá solicitar a este módulo generar una entrada en el log de GEMA proporcionando los datos necesarios.

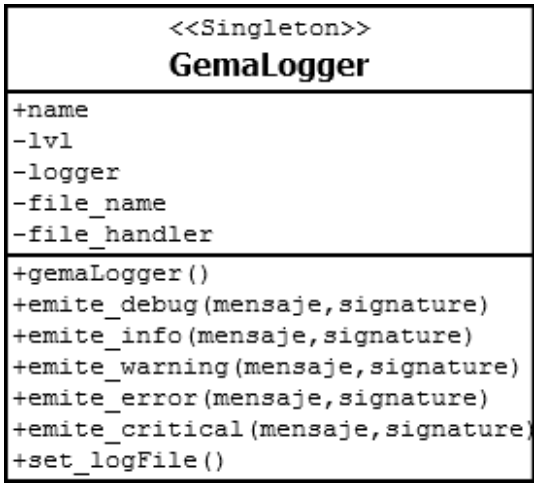


Figura 3.51. Clase GemaLogger.

Se definen cinco métodos para la emisión de entradas del log. Cada método corresponderá con uno de los niveles de relevancia definidos para los mensajes: Debug, Info, Warning, Error, Critical.

El Logger se configurará según las especificaciones del módulo de configuración. Estas especificaciones definen el path del fichero de logs, tamaño máximo del fichero, número de ficheros y el nivel mínimo de logging. El nivel mínimo permite al sistema ignorar las entradas por debajo de un umbral determinado, permitiendo así no generar entradas, por ejemplo, del nivel de debug.

El fichero de logs se define como un fichero rotatorio. Dependiendo de lo especificado en la configuración el fichero alcanzará un tamaño máximo. Una vez alcanzado este tamaño el fichero será cerrado y renombrado con un sufijo que especifique un orden para crear un nuevo fichero. Si se hubiera alcanzado el número máximo de ficheros permitido por la configuración el fichero más antiguo sería eliminado y todos los demás verían su sufijo desplazado.

3.6.8 Config

El módulo Config es el encargado de mantener disponibles todos los parámetros de configuración de la aplicación para cualquier elemento que pudiera necesitarlos. Para permitir esta disponibilidad y accesibilidad se implementa según un patrón singleton según la estrategia definida anteriormente.

Dado que Python es un lenguaje interpretado el módulo Config es en sí mismo el fichero de configuración. En él se definen directamente los valores que tendrán los diferentes atributos de la clase Config. Esta clase será la instanciada de forma única en el singleton.

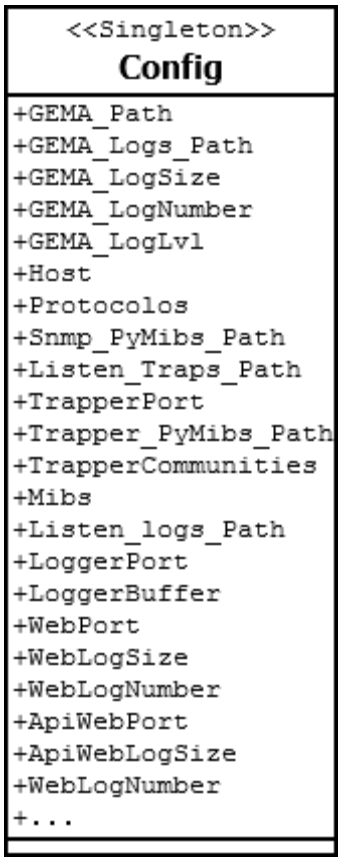


Figura 3.52. Clase Config.

De esta forma el singleton Config no dispone de ningún método al que realizar llamadas si no que todos sus atributos son leídos directamente por los diferentes elementos que requieran valores de configuración.

Entre los valores que se configuran se encuentran paths de ficheros, modos de trabajo de algunos módulos, puertos en los que habilitar los servicios web y de escucha, directorios de MIBs, etc.

3.6.9 Parser

El módulo Parser se encarga de implementar, revisar, analizar y traducir el lenguaje de programación de propiedades GEMA-Script.

El leguaje GEMA-Script, definido en profundidad en un capítulo más adelante, permite definir el comportamiento de las propiedades de GEMA. El Parser se encargará de revisar este código y traducirlo a una definición de código Python. Esta definición será proporcionada a la propiedad la cual la ejecutará al ser habilitada. A partir de ese momento la definición del método ‘Code’ queda ejecutada y el método disponible para su uso. Todo el proceso se realiza en tiempo de ejecución gracias a la capacidad de reflexión del lenguaje.

El Parser también se encarga de traducir y comprobar la corrección de las referencias a los elementos del layout en el código recibido. Sin embargo no verificará su adecuación al layout en curso. De este modo queda desacoplado del diseño actual cargado en este y trabaja de forma genérica contra la definición del diseño.

Gracias a este comportamiento es perfectamente posible modificar por completo el lenguaje de scripting de GEMA siempre que el nuevo módulo implementado respete el diseño del layout en los direccionamientos y posea el método 'parse' que devuelva el código de la definición en Python correctamente junto con las listas de referencias.

Las listas de referencias consisten en tres listas con los nombres de los elementos referenciados en el código GEMA-Script. Estas listas representarán los accesos absolutos, primer elemento de una referencia y último; los accesos por conexión, aquéllos elementos accedidos a través de un enlace por una conexión y los accesos relativos, todos los saltos intermedios en una referencia.

En la referencia "NodoC1;Hijo2;NodeA;Hijo1;var5" las listas de accesos serian las siguientes:

- Absolutos: NodoC1, NodoC1; Hijo2;NodeA;Hijo1;var5
- Por Conexión: NodoC1;Hijo2;NodeA;Hijo1
- Relativos: NodoC1;Hijo2, NodoC1;Hijo2;NodeA;Hijo1

Estas referencias serán utilizadas para verificar la integridad del layout a la hora de eliminar elementos. Estas referencias son traducidas a accesos a los elementos en Python por los dos métodos específicos diseñados, uno para los accesos de lectura y otro para los de escritura.

Debido a la traducción de estas referencias existe un acoplamiento inevitable entre este módulo y el diseño del layout. Cualquier cambio en la estructura del layout deberá implicar cambios en la implementación de las traducciones de estas referencias.

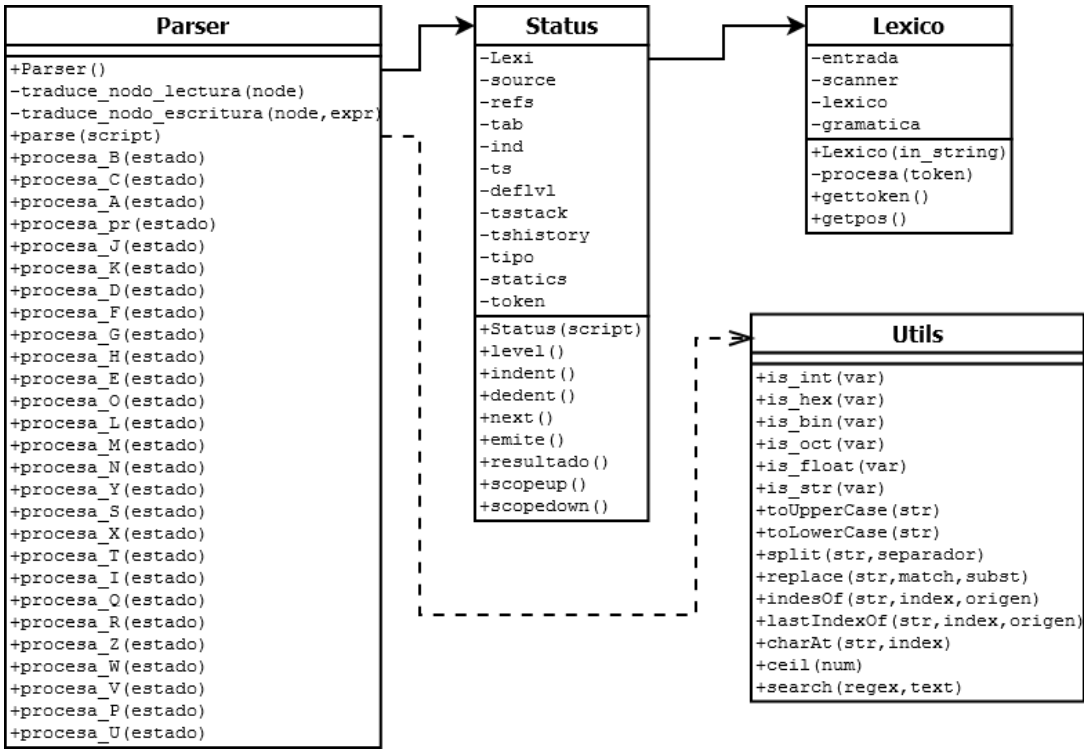


Figura 3.53. Diagrama de clases del Parser.

El Parser al completo cuenta con tres componentes diferenciados:

- Parser: Es el encargado de representar la sintaxis del lenguaje y de realizar su análisis y traducción. Su implementación se realiza mediante el método recursivo predictivo por considerarse un método flexible y fácil de modificar además de intuitivo a la hora de revisar el código, permitiendo un fácil mantenimiento y mejora del mismo.

  Cada método del Parser, exceptuando los dos métodos de traducción explicados anteriormente, se encargará de procesar y analizar una producción de la gramática. Cada uno de ellos irá llamando, recursivamente, al siguiente en función de los tokens que va recibiendo.

  Cuando se alcanza una producción terminal se escala la secuencia de llamadas recursivas devolviendo los fragmentos de código de cada producción hasta encontrar otra bifurcación en el árbol de la gramática o llegar a una producción raíz que emitirá el código generado correspondiente.

- Status: Esta clase mantiene registrado en todo momento el estado del proceso de compilación. Gracias a ella es posible realizar diferentes procesos de compilado concurrentemente, cada uno de ellos con su propio Status. La clase incluye un analizador léxico con el código precargado y su propio puntero de lectura con el último token leído, la jerarquía de tablas de símbolos necesarias para todas las definiciones, valores relativos a los niveles de indentación requeridos por el código Python, valores generales como el tipo definido para el código, la lista de referencias acumuladas por los accesos indicados en el código y el código fuente generado hasta el momento.

  Proporciona los métodos necesarios para el análisis y generación del código:

  - ‘Indent’ y ‘Dedent’ permiten modificar el nivel de tabulado actual en el código fuente generado hasta el momento.
  - ‘ScopeUp’ y ‘ScopeDown’ permiten subir o bajar el nivel de tabla de símbolos actual para el tratamiento de definiciones de función anidadas.
  - ‘Emite’ añade una línea de código.
  - ‘Next’ obtiene el siguiente token, avanzando el puntero de lectura del léxico.
  - ‘Level’ nos indica el nivel de indentación actual.
  - ‘Resultado’ devuelve el resultado actual del proceso.

    (Código generado, Referencias, Tipo de la propiedad)

- Léxico: Esta clase representa la gramática léxica del lenguaje GEMA-Script. Es la encargada del análisis léxico del código GEMA-Script. Para su implementación se apoya sobre la librería ‘Plex’ de Python.

  La clase ofrece únicamente dos métodos, ‘gettokken’ que devuelve el siguiente token del el código y avanza el puntero de lectura y ‘getpos’ que devuelve la posición actual del puntero dentro del código.

Internamente define los diferentes tokens de los que consta el lenguaje. Estas definiciones, junto con la definición de los diferentes tipos de token que se utilizarán, son utilizadas para construir un analizador léxico con la librería 'Plex'. Cada token devuelto por el analizador de la librería es procesado por el método 'Procesa' para adecuarlo al formato esperado por el analizador sintáctico 'Parser'.

La clase Utils incorpora una serie de definiciones de función estándar. Este modulo es referenciado e importado desde el código de las propiedades. La importación se realiza de forma absoluta, esto implica que los nombres definidos en el pasan a formar parte del modulo que lo importa. De esta forma se evita tener que replicar este código en la cabecera fija de cada una de ellas quedando las funciones disponibles directamente en su código.

Una vez finalizado el proceso de traducción el Parser devolverá la terna de resultados (Código generado, Referencias, Tipo de la propiedad) o el código de error correspondiente si se hubiera producido alguno.

Este código traducido a Python en ningún momento sale de la estructura interna de GEMA. Al ser código que se va a ejecutar internamente es preciso asegurar que no pueda ser manipulado externamente para evitar cualquier posibilidad de inyección de código en el sistema. De cara al exterior el método 'parse' proporcionado por el API únicamente devolverá un valor cierto si la traducción se ha realizado con éxito o una instancia de Error representando el error acontecido durante el proceso.

3.7 Diseño e implementación de la capa de procesos

Este nivel agrupa a las diferentes clases activas que componen la aplicación. Estas clases, a diferencia de las clases normales descritas anteriormente, implican un hilo de ejecución paralelo a todos los demás. Estas clases serán las responsables de simultanear y coordinar la ejecución de las diferentes funcionalidades de GEMA.

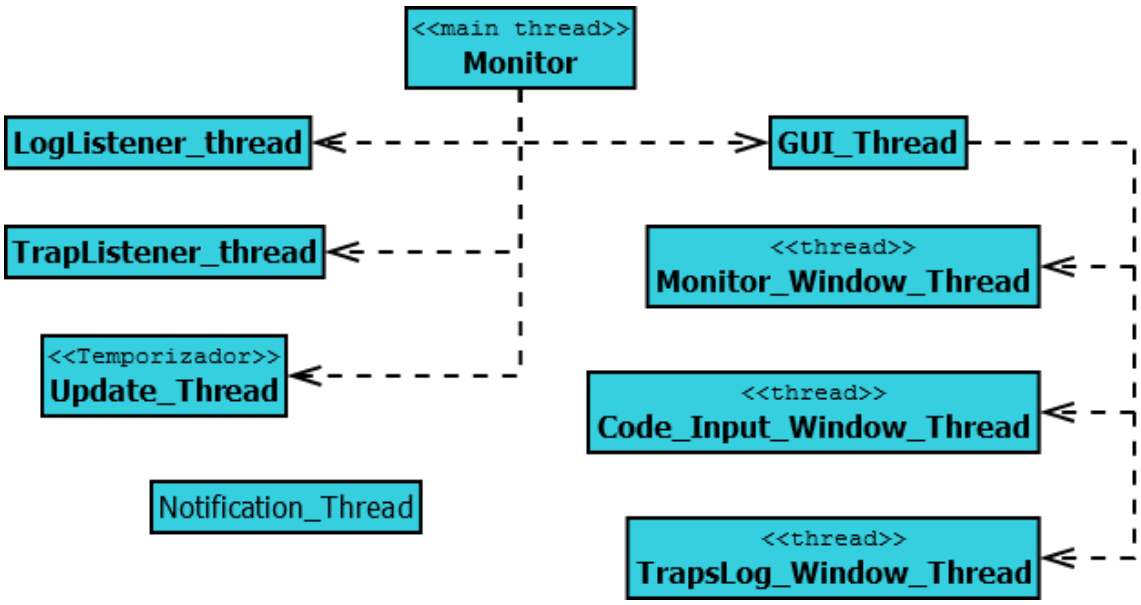


Figura 3.54. Modelo simplificado de clases de la capa de procesos.

Para la división de la aplicación en threads ha sido necesario evaluar los posibles efectos negativos de la concurrencia. Se ha verificado que la mayoría de accesos a los recursos compartidos de GEMA son de lectura y que únicamente habrá un thread que acceda en modo de escritura a estos elementos. Las variables únicamente serán escritas por el thread de actualización interno a los protocolos o, en su defecto, por el thread de update de variables de su propio nodo. Por su parte, las propiedades únicamente pueden ser escritas por sí mismas.

Los posibles entrelazados entre lecturas y escrituras no suponen un riesgo grande en el funcionamiento de la aplicación. Únicamente podrán implicar una desactualización de un ciclo en los valores de las variables o propiedades y salvo picos instantáneos en los valores el retardo provocado no es relevante. Este hecho debe ser tenido en cuenta por los diseñadores del Layout y programadores de GEMA-Script a la hora de establecer los intervalos de los diferentes temporizadores o de obtener el valor de los diferentes elementos, pues estos pueden ser actualizados durante la ejecución del código de la propiedad.

Para los posibles problemas provocados por el diseño concurrente de la red, dos administradores modificándola simultáneamente, el API incorpora un sistema interno de cerrojos que previenen la eliminación de elementos durante las operaciones de actualización de las propiedades.

Esto impide posibles fallos en las referencias de las propiedades. Evita que las referencias de una propiedad se evalúen como correctas y sin embargo los elementos referenciados sean eliminados antes de ser marcados como referenciados durante el proceso de activación de la propiedad.

Para reducir el nivel de multiprogramación de GEMA los threads únicamente serán creados cuando sean necesarios y se destruirán en cuanto dejen de serlo. En los nodos, por ejemplo, únicamente existirá el thread de actualización de propiedades si existe alguna de ellas dada de alta y está activado su refresco.

Cabe destacar que el intérprete estándar de Python, CPython, incorpora un mecanismo que evita la ejecución en paralelo de los threads del mismo programa [31]. Este mecanismo, el Global Interpreter Lock (GIL) funciona como un mutex y su existencia se debe a que la gestión de memoria interna del intérprete no es “thread-safe”.

Si bien esto puede suponer una pérdida de eficiencia potencial en sistemas multiprocesador, el GIL no impide la ejecución paralela de procesos. Dado que nunca habrá más de un thread en ejecución en un procesador concreto se recomienda, para estos casos, la paralelización del código en tantos procesos cono núcleos posea el sistema y utilizar dentro de ellos los threads que sean necesarios.

También se ha de tener en cuenta que, durante las operaciones de entrada y salida así como operaciones de alta carga de computación, el GIL puede ser liberado por los módulos, dejando así el mutex disponible para la ejecución de otro thread durante su proceso.

Este mecanismo no debe ser tenido en cuenta como un sistema de seguridad ante la concurrencia, pues cualquier efecto negativo de la misma puede darse incluso en sistemas de un solo núcleo debido a los cambios de contexto entre los threads o procesos en ejecución.

Existen alternativas de intérprete cuya implementación no contiene el GIL, Jyton, IronPython o StacklessPython por ejemplo, sin embargo su compatibilidad no se puede asegurar al 100%, principalmente con librerías importadas de C. La mayor parte de estas librerías se apoyan en las garantías que la existencia del GIL confiere, asumiendo que cuando están en ejecución el resto de threads están bloqueados.

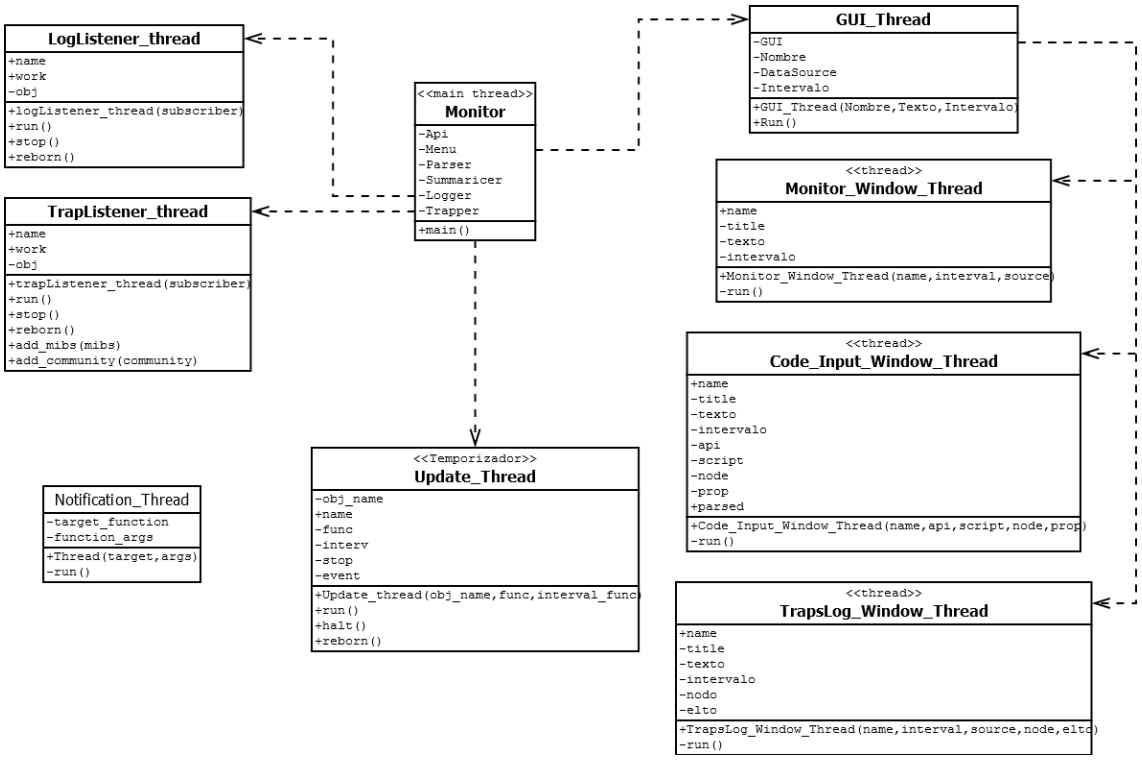


Figura 3.55. Diagrama de clases del nivel de procesos reducido.

En Python los threads son objetos y los diferentes threads de GEMA estarán referenciados por los objetos que los crearon. Una excepción a esto son los threads de notificación. Los threads de las notificaciones son creados de forma anónima y por medio de referencias a funciones les es asignado su trabajo. Estos threads anónimos desaparecerán inmediatamente tras terminar la ejecución requerida por las notificaciones.

En la figura 3.55 las referencias indicadas entre los threads no son directas, pues se omiten los objetos de otras capas intermedios en estas relaciones. Simplemente muestra una jerarquía de ejecución de los threads indicando que thread sería inmediatamente superior a cada uno, sin embargo, al ser los threads objetos independientes, esta información no es vinculante.

Podría desaparecer el GUI_Thread y sin embargo seguir existiendo y trabajando los threads de cada una de sus ventanas. Si bien esto provocaría un malfuncionamiento del interfaz al no responder el thread de la GUI a sus peticiones el funcionamiento global de GEMA no se vería afectado y cada una de las ventanas seguiría refrescando su información perfectamente.

A la hora de programar propiedades asíncronas, y en general cualquier propiedad, se deben evitar los bucles infinitos de espera, tanto activa como pasiva.

Las propiedades asíncronas deberán resolver su ejecución en una sola iteración finita y las propiedades síncronas tendrán en cuenta la existencia de un thread de update síncrono que las ejecutará cada cierto intervalo de tiempo. Podrán ambas almacenar en valores estáticos internos aquella información que sea necesaria recordar de anteriores ejecuciones.

A continuación se describe más en detalle cada una de estas clases indicando su funcionalidad y aspectos destacables de su implementación.

3.7.1 Main (Monitor)

Es el hilo principal de ejecución arrancado directamente desde la consola de sistema. Desde el se crean los demás componentes iniciales de GEMA y lanza la ejecución del menú de consola.

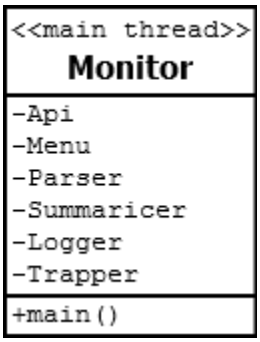


Figura 3.56. Clase Monitor (Main thread).

Este thread creará las diferentes instancias de las clases principales y las referenciará entre sí. Es también responsable de inicializar todos los singleton de GEMA.

El thread Main es la raíz de toda la jerarquía de threads y su interrupción abortaría la ejecución completa de la aplicación. También es el responsable de mantener referencia al estado global de la máquina, utilizado por el terminal interactivo de Python para permitir el acceso a todos los subelementos. Por esta razón hay que ser especialmente precavidos en el uso de este terminal para evitar abortar la ejecución de toda la aplicación.

3.7.2 Threads notificaciones

Cada notificación generada en GEMA se lanza directamente en sus propios threads dedicados. Esto permite la inmediata atención a la notificación por el método solicitado sin tener que encolar su trabajo dentro de otro hilo de ejecución. De esta forma se consigue un tratamiento totalmente asíncrono de las notificaciones.

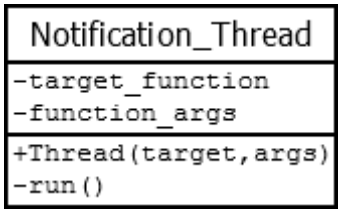


Figura 3.57. Clase Thread Notificación.

Por cada subscriptor destinatario de la notificación se crea un thread anónimo al que se le pasa como función a ejecutar una referencia al método de notificación apropiado del destinatario.

Una vez resuelto el tratamiento de la notificación por el destinatario el thread es eliminado liberando así los recursos del sistema y aligerando la carga del planificador.

Dentro de la carga de trabajo de estos threads se incluye tanto la transmisión a subscriptores remotos como la ejecución de las propiedades asíncronas. Una misma propiedad asíncrona puede estar en ejecución en más de un thread. Esto ocurrirá cuando dos notificaciones deban ser atendidas por la misma propiedad. La ejecución de la propiedad se realizará de forma concurrente.

Esta posibilidad de ejecución concurrente deberá tenerse en cuanta durante su programación a diferencia de las propiedades síncronas. Las propiedades síncronas únicamente son ejecutadas por su thread de update y por tanto no existe posibilidad de ejecución concurrente de una misma propiedad.

Las propiedades asíncronas pueden estar ejecutándose concurrentemente y esto implica que sus valores estáticos pueden ser modificados simultáneamente. Por tanto no se puede asumir una acumulación o modificación secuencial de estos valores. Una ejecución más antigua puede terminar su trabajo después de una ejecución activada por una notificación posterior. Por ello es recomendable programarlas de la forma más independiente de ejecuciones anteriores posible, o, como mínimo, no asumir linealidad en las mismas.

Se ha de ser especialmente precavido en el encadenado de llamadas provocadas por mensajes. Si bien el sistema impide que un mensaje generado por una propiedad pueda activar su propia ejecución no impide un encadenamiento de llamadas a través de una segunda propiedad. Esto es que la Propiedad asíncrona A1 genera un mensaje que despierta la propiedad A2 y esta a su vez genera uno que despierte a A1 nuevamente.

Esta situación provocaría la ejecución constante y descontrolada de estas dos propiedades. Si bien esta ejecución es asíncrona y cada thread moriría tras despertar la siguiente propiedad no aumentando el nivel de carga del sistema.

Más grave sería si cualquiera de las dos propiedades fuese capaz de despertar a otras dos. En esta situación la ejecución descontrolada de estas propiedades provocaría una reacción en cadena que dispararía el nivel de multiprogramación del sistema consumiendo todos los recursos de ejecución dejando la máquina sin capacidad de proceso.

Dada la complejidad de entrelazado de llamadas que puede llegar a implementarse GEMA no es capaz de predecir esta situación. Es responsabilidad de los administradores que programen las propiedades hacer un correcto estudio y uso de los mensajes generados y las expresiones regulares que determinan la activación de las propiedades.

El encadenamiento de llamadas es una herramienta útil a la hora de evitar la replicación de código y modularizar la programación de GEMA-Script, pero debe hacerse con el debido estudio y análisis para evitar este tipo de situaciones.

3.7.3 Threads listeners

Estos threads mantienen los servicios de escucha en los puertos definidos para la captura de los Traps y los Logs. Son los encargados de dotar de independencia y asincronismo a los listeners de GEMA: El Trap Listener y el Log Listener.

Log Listener Thread: El thread del LogListener se encarga de la creación y activación del socket de escucha de los Logs de la red.

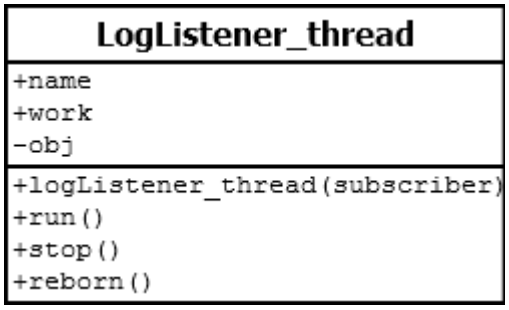


Figura 3.58. Clase Log Listener Thread.

El thread controla la actividad del listener, "obj". Una vez creado el socket el listener queda a la espera de un Log. Una vez capturado y procesado el control vuelve al thread quien evalúa si se ha solicitado la detención del servicio. Si el servicio no ha sido detenido se reactiva el listener que queda a la espera de un nuevo Log. En caso de una petición de detención el thread se restaura al estado inicial mediante la invocación de su propio constructor.

Para la detención del servicio, el método "stop" marca a falso el flag "work" y envía una solicitud de apagado al listener, de forma que este finaliza su ejecución sin necesidad de esperar a la llegada de un nuevo Log antes de devolver el control al thread.

Trap Listener Thread: Es el encargado de mantener el servicio de escucha de traps snmp mediante los servicios proporcionados por la librería PySNMP.

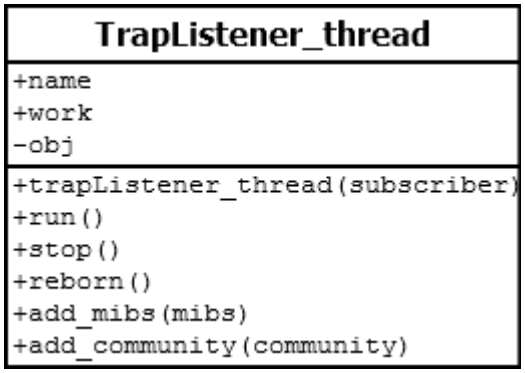


Figura 3.59. Clase Trap Listener Thread.

El funcionamiento es similar al thread del LogListener, una vez creado es activado el Listener de los Traps dentro del bucle de control. A diferencia del LogListener el TrapListener no devolverá el control al thread cada trap capturado. El listener continuará capturando y notificando los traps recibidos de forma continua.

En caso de un fallo en la recepción o funcionamiento del listener este se detendrá y devolverá el control al thread generando una excepción. La excepción será tratada y en caso de que no se haya solicitado la detención completa del thread se volverá a iniciar el servicio de escucha.

Para la detención del TrapListener se utiliza el mismo mecanismo que con el LogListener. Se detiene inmediatamente el servicio devolviéndose el control al thread el cual finaliza su ejecución y queda listo para su reactivación tras la ejecución de su propio constructor.

3.7.4 Threads GUIs

Estos threads mantienen las ventanas del interfaz gráfico local. Son los encargados de dotar de independencia y asincronismo a las ventanas del interfaz gráfico local del menú de GEMA.

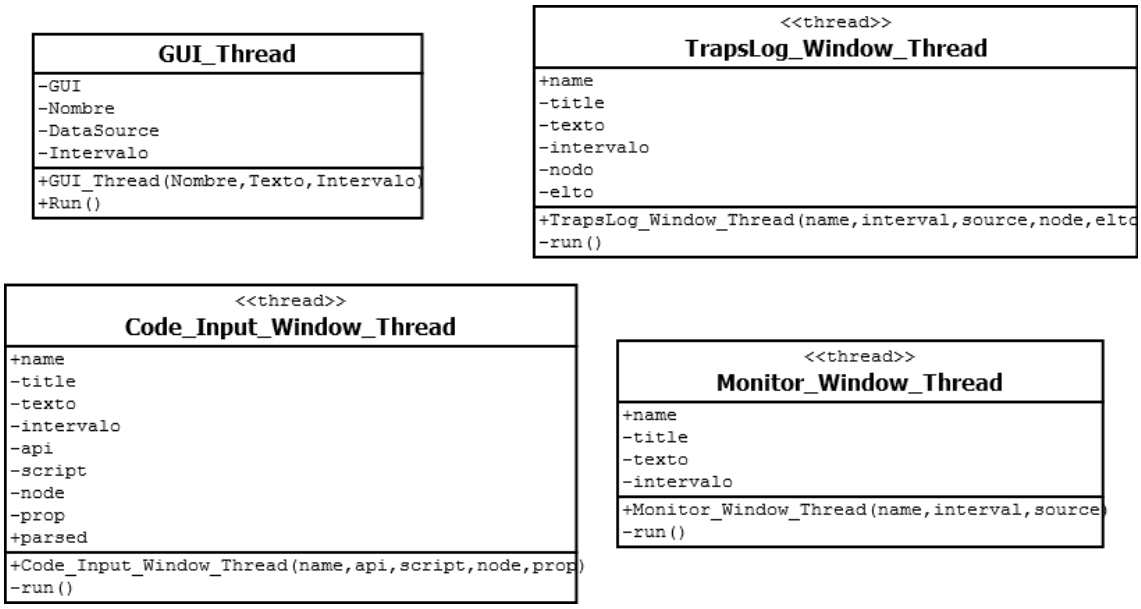


Figura 3.60. Diagrama de clases Window Thread.

Estos threads son específicos para cada instancia de ventana, creándose en el mismo instante de aparición de la misma y destruyéndose con ella. Así la carga de multiejecución debida al interfaz es proporcional al número de ventanas abiertas. El GUI_Thread coordina la creación de ventanas e independiza la visualización del thread principal del menú de consola, evitándose así la propagación de fallos en el interfaz gráfico.

TrapsLog_Window_Thread mantiene la actualización de una ventana que monitoriza y muestra el valor de un elemento de un nodo especificado. Originalmente diseñados para monitorizar los Traps y Logs de un nodo, su implementación permite monitorizar cualquier componente de un Nodo a través del la función “view_elto_data” del API.

Monitor_Window_Thread, al igual que el TrapsLog, actualiza y monitoriza el Layout completo mostrándolo en la ventana en una representación jerárquica arborescente.

Code_Input_Window_Thread mantiene una ventana de edición de código de propiedades sencilla. Esta ventana, inicializada con el código actual cargado en la propiedad, permite editar la programación de las propiedades.

Al confirmar el código introducido se llamará directamente al servicio de edición de propiedades del API pasando el código como parámetro. En función del resultado devuelto por el API se notificará al usuario del resultado. En caso de que se detectasen errores en el código se mostrará una ventana emergente indicando el error que abortó la compilación o activación de la propiedad.

De esta forma el editor no tiene acceso en ningún momento al Parser directamente ni al código Python generado. Únicamente recibirá los resultados de la compilación por mediación de la API.

3.7.5 Updater

La clase Updater es una especificación de la clase genérica de Python "Thread". Esta clase Thread implementa un tipo de objeto especial caracterizado por contener un método, "run" que puede ser lanzado en un hilo de ejecución independiente. Esto permite en Python que los threads sean tratados como un objeto más dentro del código, lo que simplifica su implementación al tiempo que les da una enorme flexibilidad al poder ser extendidos mediante la herencia y el polimorfismo propios de los objetos.

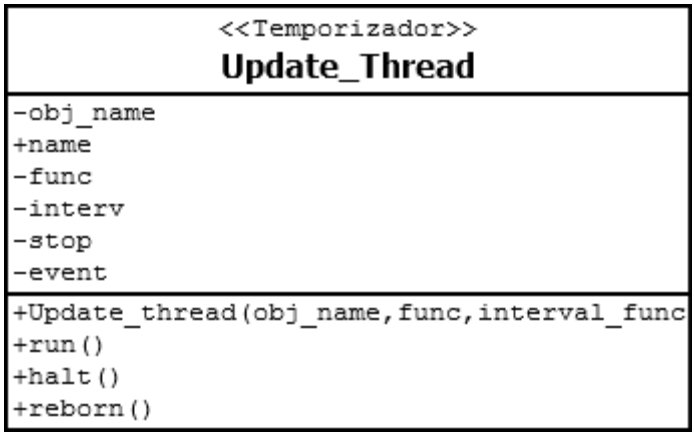


Figura 3.61. Clase Updater.

En la implementación del Updater se extiende a Thread añadiéndole referencias a la función específica del objeto que ha de actualizar y a la función que indicará en cada iteración el intervalo de espera. Estas referencias son específicas a una función dentro de un objeto concreto y no a la función genérica dentro de una clase. Gracias a estas referencias el Updater se abstrae completamente de la clase que lo utilice o del código que ejecutará dentro de su hilo, permitiendo ser utilizado por cualquier módulo que requiera programar la ejecución de una función tras un intervalo de tiempo dado.

La implementación también dota al Updater de métodos capaces de detener y reanudar la ejecución, funcionalidad no incorporada por los thread de serie de Python. Para conseguir esto el propio thread, justo en el instante anterior a su muerte, invoca a su propio constructor padre quedando de nuevo en el estado de no activado sin que esto provoque ningún cambio en los elementos añadidos en su especificación en la clase Updater.

De esta forma el Updater es capaz de mantener memoria de su estado en la anterior ejecución sin que sea necesario crear uno nuevo tras cada parada.

Para implementar el control y sincronización de las llamadas a los métodos de la clase y evitar efectos indeseados por llamadas en rebote (llamadas repetidas, por ejemplo, a la parada o arranque del thread) se utiliza la clase "Event" de Python que permite a una ejecución quedarse esperando la llegada de un evento. Controlando el armado y desarmado de esta señal se consigue controlar la concurrencia de las solicitudes y atenderlas de forma asíncrona. De esta forma el proceso solicitante de la parada del thread no queda esperando a que esta se efectúe antes de poder continuar con su trabajo.

Dado que cada propiedad o variable pertenecen a un único "Elto" y este a su vez posee un único Updater dedicado a ellas no se observa necesario implementar un control de concurrencia ante la eventualidad de que dos hilos solicitasen simultáneamente la actualización. Esto ha de ser tenido en cuenta y respetado por cualquier módulo que desee utilizar esta clase como cronómetro para disparar una función.

3.8 Diseño e implementación del lenguaje de scripting Gema Script

En esta sección describiremos el diseño e implementación del lenguaje de programación de propiedades GEMA-Script. Se ha diseñado un lenguaje sencillo para la descripción de propiedades partiendo de Java Script. Se ha omitido la abstracción de objetos orientando el lenguaje a un paradigma meramente imperativo.

Algunas decisiones de diseño se han basado en el hecho de que el lenguaje será traducido directamente a Python y la construcción de algunas estructuras y sentencias se han diseñado para facilitar este proceso.

Para su implementación se opta por un analizador descendente recursivo predictivo que nos permite implementar el compilador de una forma bastante intuitiva y flexible al tiempo que fácilmente mantenible y revisable por alguien que no conozca en profundidad otros métodos utilizados en la implementación de compiladores.

Este método implementa la gramática del sintáctico de forma directa, traduciendo cada producción del lenguaje en una función. Cada una de estas funciones detectará los símbolos que le llegan en forma de tokens del lenguaje y determinará, por medio de condicionales, la siguiente producción a evaluar. Una vez determinada llamará a la función correspondiente a esa producción. Cada función irá acumulando los fragmentos de código intermedio, Python, durante el proceso recursivo de llamadas, tanto durante el descenso como durante el ascenso según corresponda. Una vez alcanzado el nivel de producción de una sentencia tras su evaluación recursiva se emitirá el código acumulado correspondiente.

De esta forma cualquier cambio en la gramática, modificación o adición, simplemente deberá traducirse en la modificación del código de las funciones-producción implicadas y el añadido de las nuevas si fuera el caso.

La lectura conjunta del código del Parser junto con la gramática definida permitirá comprender fácilmente el funcionamiento del Parser y la gramática implementada.

3.8.1 Descripción del lenguaje

A continuación se describen las características del lenguaje diseñado. Se describirán sus principales características, tipos, sentencias, sintaxis de las declaraciones, etc.

Generales:

El lenguaje es insensible a mayúsculas y minúsculas en sus palabras reservadas. Si distinguirá en los nombres de variables y funciones declaradas por el programador así como en lo "paths" a elementos de GEMA.

Declaración del tipo de la propiedad:

Con la directiva "#%TYPE:Tipo" declaramos el tipo que se asignará a la propiedad:

#%TYPE:Entero

El tipo es un valor de la propiedad e independiente de la ejecución del código, se quedará asignado el último especificado en el código en caso de aparecer la directiva repetida.

Su uso está destinado a su evaluación por otras propiedades y, principalmente, al interfaz gráfico a la hora de determinar su representación adecuada.

Declaración de constantes:

La declaración de los valores constantes se realizará según el siguiente esquema en función de su tipo:

Booleanos, literales:	*true / false*
Enteros, se indicará directamente el número:	*7534*
Reales, se indicaran por la presencia del punto decimal:	*7324.0241*
Hexadecimales, se indicarán encabezados por 0x:	*0x08C4A467DF*
Binarios, se indicarán encabezados por 0b:	*0b00101101010*
Octales, se indicarán encabezados por 0:	*0254336372*
Cadenas de texto, se indicaran entre comillas dobles:	*"Lorem ipsum"*

Notas:

- Todos los tipos numéricos en diferente base serán 'casteados' internamente a enteros.
- Para indicar una comilla doble en una cadena esta deberá 'escaparse' doblemente:

  "Esta cadena contiene una \\" comilla doble"

- Indicar un número encabezado a cero fuera del rango octal producirá un error:

  *a = 021269 → **ERROR**. El 9 no pertenece al rango octal.*

- Se podrá anteponer un "r" a la declaración de una cadena para evitar que se interpreten caracteres de escape:

  A = r"esta cadena NO contiene \n un salto de línea"

Declaración implícita de variables:

variable_no_declarada = Expresión

Nótese que los nombres de variable deberán comenzar estrictamente por una letra y podrán contener exclusivamente letras, dígitos y el carácter guión bajo: "_"

Vectores:

variable_vector = [0,0,0,0,0]

Los vectores son de dimensiones constantes y encabezados a cero.

El direccionamiento dentro de un vector se hará entre corchetes:

variable = variable_vector[índice]

Multidimensión en vectores:

variable_matriz_3x3 = [[0,0,0],[0,0,0],[0,0,0]]

El direccionamiento dentro de un vector se hará entre corchetes sencillos indicando los valores de direccionamiento:

variable = variable_matriz_3x3[2,1]

Nótese que los vectores se encabezan a 0:

variable_matriz_3x3[3,3] → ***ERROR.*** *Fallo de direccionamiento en ejecución.*

Variables estáticas:

static variable_no_declarada_previamente = valor_inicial_constante

Las variables así declaradas conservarán su valor de una ejecución a otra de la propiedad, deberán ser declaradas con un valor inicial constante, incluidos vectores:

static buffer = [0,0,0,0,0,0,0,0,0,0]

De esta forma creamos un buffer de memoria que, utilizado convenientemente, almacenará el historial de la propiedad.

Nótese que esto no es una sentencia a ejecutar, es una declaración independiente de donde aparezca en el código. Esta declaración será procesada y ejecutada siempre durante el proceso de compilación.

Se recomienda declara todas las variables estáticas al comienzo del código para evitar confusiones al programador.

Casteo de variables:

Se ofrecen las siguientes funciones: "int", "hex", "bin", "oct", "float" y "str" para realizar el casteo de las variables.

Las funciones “hex”, “bin”, “oct” y “str” devolverán una cadena de texto representando el valor:

hex(16) → *"0x10"*

bin(16) → *"0b10000"*

oct(16) → *"020"*

str(16) → *"16"*

La función “int” recibe un valor numérico o “cadena que lo represente en base 10” y devuelve el valor entero. Si la cadena no está representada en base 10 podrá indicarse la base o introducir un cero como segundo parámetro para que el sistema trate de deducirla:

int("16") → *16*

int(0x16) → *22*

int("0x16") → ***ERROR**. La cadena no lo representa en decimal.*

int("0x16",16) → *22 (Indicada la base explícitamente)*

int("0x16",0) → *22 (El sistema averigua la base)*

La función “int” podrá utilizarse para obtener la parte entera de un real:

int(9.8) → *9*

La función “float” recibe un valor numérico o “cadena que lo represente en base 10” y devuelve el valor real (“float” no acepta indicación de la base):

float(16) → *16.0*

float("16") → *16.0*

float(0x16) → *22.0*

float("0x16") → ***ERROR**. La cadena no lo representa en decimal.*

float(int(0x16)) → *22.0*

Declaración de funciones:

```
def Nombre_funcion(parametro1,prametro2,...)
    Bloque de sentencias...
    return variable
end
```

Las variables que se declaren dentro de una función serán visibles únicamente dentro de esta:

```
var = 0
def Nombre_funcion()
    return var
end
```

Producirá un error al no estar "var" declara dentro de la propia función. No existen variables globales ni se propaga el ámbito de las variables hacia el interior.

Anidamiento de funciones:

```
def Nombre_funcion1(parametro1,prametro2,...)
    Bloque de sentencias...
    varF1 = 10
    def Nombre_funcion2(parametro1,prametro2,...)
        Bloque de sentencias...
    end
    Bloque de sentencias...
    return variable
end
```

Nótese que las variables no se propagan hacia el interior, en la función1 se declaro la variable "varF1" pero esta no será visible desde la función2. Deberá pasarse como parámetro de ser necesaria.

Las variables estáticas únicamente existen a nivel del cuerpo principal del código. De ser requeridas en las funciones internas deberán pasarse como parámetros.

Recursividad:

```
def Nombre_función(parametro1,prametro2)
    Bloque de sentencias...
    variable = Nombre_funcion(parametro1+1,parametro2*2)
    Bloque de sentencias...
    return variable
end
```

← ***Recursión***.

Se deberán implementar los habituales mecanismos de control de la recursividad tales como casos base, casos finales, etc. para evitar un encadenamiento infinito de llamadas.

Notación direccionamiento de elementos de GEMA:

Los elementos de GEMA podrán ser direccionados para su lectura, procesado y escritura, si esta está permitida, utilizando la siguiente sintaxis de direccionamiento. Estas sentencias se denominarán "path" dentro del código de GEMA-Script.

*Nombre_Nodo;[Alias_Hijo;]**Nombre_Variable/Componente*

Deberá respetarse un espacio al inicio y final del mismo a fin de evitar ambigüedad en la interpretación del direccionamiento:

int(Nodo;Hijo;Variable)

Mediante esta notación se podrán referenciar los diferentes elementos del Layout de GEMA a los que se desee acceder. El último elemento "Nombre_Variable/Componente" podrá se cualquier atributo legible de los nodos o conexiones además de sus variables y propiedades.

Para los componentes que dispongan de un tipo declarado se podrá añadir un último elemento ";__tipo__" para direccionarlo.

int(Nodo;Hijo;Variable;__tipo__)

Para acceder a los nodos de una conexión se utilizarán los alias "ExtrA" y "ExtrB".

int(Conexion;ExtrA;Variable;__tipo__)

Set y Get de variables de los nodos:

La asignación de valores, "set", a variables de un nodo se realizará como cualquier operación de asignación:

Nodo;Hijo;Variable = Expresión

Esta operación desencadenará una operación "SET" sobre la variable mediante el protocolo correspondiente.

Para la lectura, "get" de variables y cualquier atributo legible del los nodos se utilizará su direccionamiento en el formato "path" indicado anteriormente:

variable = funcion(int(Nodo;Hijo;Variable))

Nótese que esto devolverá el último valor leído o actualizado de la variable, propiedad o atributo. Esta sentencia no desencadenará ninguna actualización u operación "GET" sobre el protocolo. Estos valores son únicamente actualizados por los diferentes mecanismos de actualización, threads, de GEMA.

Deberá tenerse especial cuidado a la hora de leer varias veces el mismo elemento durante la ejecución de la propiedad, pues su valor podría cambiar a mitad de la ejecución debido al asincronismo de los threads de actualización. Se recomienda hacer una única lectura al inicio del código y almacenarla en una variable a la que se hará referencia en el resto de las sentencias para evitar esta contingencia debida a la concurrencia.

Condicionales:

Las sentencias condicionales se especificarán siempre por bloques con una cláusula de cierre en cada uno de ellos:

```
if Expresión
    Bloque de sentencias...
elif Expresión
    Bloque de sentencias...
else
    Bloque de sentencias...
end
```

La sentencia "*elif*" junto con su bloque de sentencias correspondiente podrá repetirse las veces que sean necesarias. La sentencia "*else*" podrá aparecer como máximo una vez en el código.

Por medio de la concatenación de las sentencias "*elif*" se implementará el comportamiento de la estructura condicional "*select*". Tanto la sentencia "*elif*" como "*else*" son opcionales así como sus bloques.

Bucles:

Se proporcionan las estructuras de control de bucles "*For*" y "*While*" con la siguiente sintaxis:

```
while Expresión
    Bloque de sentencias...
end
for variable in Expresión
    Bloque de sentencias...
end
```

La expresión del "*For*" deberá devolver un objeto iterable (cadena de texto o vector)

Operadores definidos:

Se proporcionan los siguientes operadores para la construcción de expresiones:

- Operadores matemáticos: suma +, resta -, división /, producto * y módulo %.
- Operadores lógicos: negación !, conjunción &&, disyunción ||.
- Operadores relacionales: mayor o igual >=, menor o igual <=, igual ==, distinto !=, mayor >, menor <)
- Operadores a nivel de bit: AND &, OR |, complemento ~, XOR ^
- Operadores de desplazamiento: a la izquierda<<, a la derecha >>

Funciones predefinidas:

A continuación se describen las funciones predefinidas en el lenguaje para su uso directo dentro del código de las propiedades. Como ya se indicó anteriormente estas funciones vienen declaradas dentro del módulo Utils de la implementación del Parser cuando no sean funciones básicas de Python directamente permitidas en la definición del léxico.

- Generadores:
    - range(A,B) devuelve un vector con los enteros desde 'A' a 'B' (no incluido)

      range(0,9) → *[0,1,2,3,4,5,6,7,8]*

      De no indicarse el origen este se considerará 0

      range(9) → *[0,1,2,3,4,5,6,7,8]*

      Podrá especificarse el valor de cada paso:

      range(9,20,2) → *[9, 11, 13, 15, 17, 19]*

      range(9,0,-2) → *[9, 7, 5, 3, 1]*

Será especialmente útil en la implementación de bucles "For"

```
for variable in range(valor_inicial, valor_final, valor_paso)
    Bloque de sentencias...
end
```

- random() devuelve un valor aleatorio real en el intervalo [0 y 1)

  random() → *0.9845597907344845*

- Timestamps:
  - timestamp() devolverá la hora y fecha en una cadena de texto:

    timestamp() → *"Thu Jan 24 12:51:06 2013"*

  - time() Devuelve el número de segundos desde epoch, 1 de Enero de 1970, la precisión depende exclusivamente de la máquina.
  - ctime(secs) Devuelve el 'timestamp' de la fecha secs expresada en segundos desde epoch. Si se omite el parámetro "secs" utilizará la fecha actual, resultado equivalente a timestamp()
- Comprobaciones de tipo:
  - Se implementan las siguientes funciones, que devolverán un valor booleano, para verificar si el casteo de tipos sería válido para el parámetro dado:

    is_int(var), is_hex(var), is_bin(var), is_oct(var), is_float(var), is_str(var).
- Funciones de formateo de cadenas:
  - toUpperCase(cad) Devolverá toda la cadena en mayúsculas.
  - toLowerCase(cad) Devolverá toda la cadena en minúsculas.
  - split(cad, sep) Devolverá un vector de cadenas separando la cadena original por el carácter separador indicado.
  - substr(var,ind,fin) Devuelve la subcadena de "var" entre las posiciones "ind" y "fin" (no incluida) substr("hola", 1,3) → "ol")
  - replace(cad, ori, fin) Devolverá la cadena resultado de reemplazar la cadena "ori" en la cadena "cad" por la cadena "fin".
  - indexOf(cad, var, ori) Devuelve la primera posición en la que aparece "var" en "cad" a partir de origen indicado "ori".
  - lastIndexOf(cad, var, ori) Devuelve la última posición en la que aparece "var" en "cad" a partir de origen indicado "ori".
  - charAt(cad, ind) Devolverá el carácter en la posición indicada.

- Funciones matemáticas:
    - ceil(num) Devuelve el entero inmediatamente superior al número real dado.
    - len(var) Devuelve la longitud de un vector o cadena de texto.
    - round(var) Devuelve el valor entero más próximo a la variable.
    - pow(var,pot) Devuelve la potencia "pot" de "var".
    - exp(var) Devuelve e elevado a "var".
    - log(var,B) Devuelve el logaritmo en base 'B' de "var", si no se especifica la base se asume base e, logaritmo neperiano.
    - sqrt(var) Devuelve la raíz cuadrada de var.
    - max(var) Devuelve el mayor de sus argumentos o el mayor elemento del un solo argumento iterable (cadena de texto o vector)
    - min(var) Devuelve el menor de sus argumentos o el menor elemento del un solo argumento iterable (cadena de texto o vector)
    - abs(X) devuelve el valor absoluto de X:

        abs(-89.98) → *89.98*

- Expresiones regulares:
    - search(regex, text) Indicará si "text" cumple la expresión regular indicada.

3.8.2 Gramática del léxico

En esta sección se describe la gramática del analizador léxico. Se presentan los diferentes tokens que se definen para GEMA-Script. Para la definición de estos token se define una serie de categorías y en base a estas se construye la definición de los token.

- Dígito: Cualquier valor entre 1 y 9 ambos incluidos.
- Dígito no cero: Cualquier valor entre 1 y 9 ambos incluidos.
- Letra: Cualquier carácter alfabético.
- Nombre: Cualquier letra seguida de una secuencia de letras, dígitos o el carácter "_".
- Path: Cualquier secuencia de cadenas de letras, dígitos o los caracteres "_ : - ." concatenadas por ";".
- Int: Cualquier digito no cero seguido de una secuencia de dígitos o un solo "0".
- Binary: Cualquier secuencia de "0" o "1" encabezada por "0b".
- Boolean: Cualquier "true" o "false" independiente de mayúsculas.
- Octal: Cualquier secuencia de los caracteres "0 1 2 3 4 5 6 7 8" encabezada por un "0".
- Real: Cualquier Int seguido del carácter "." Y una secuencia de dígitos.

- Hexadecimal: Cualquier secuencia de los caracteres "0 1 2 3 4 5 6 7 8 9 A B C D E F " encabezada por “0x”
- Cadena: Cualquier secuencia de caracteres encabezada por “ o r” y terminada por “.
- OperadorArit1: "+” o “-"
- OperadorArit2: Cualquiera de "/”, “*”, “%”
- OperadorLog: Cualquiera de “!”, “&&”, “||”
- OperadorRel: Cualquiera de “>=”, “<=”, “==”, “!=” “>”, “<”
- OperadorLogBit: Cualquiera de “&”, “|”, “~”, “^”
- OperadorDesBit: “<<” o “>>”
- Bloque: Cualquiera de “elsif”, “else”, “end”, “in”
- Reservada: Cualquiera de “def”, “if”, “for”, “return”, “static”, “while”
- Blacklisted: Cualquiera de “print”, “import”, “exec”, “from”

Esta última categoría, Blacklisted, define una serie de palabras que no deberán aparecer bajo ningún concepto en el código de un Script para evitar inyección de código externo o ejecución de código no controlado o evaluado.

- Directiva: Cualquier secuencia de caracteres encabezada por “#%” y terminada en “\n”, “Fin de línea” o “Fin de fichero”.
- Comentario: Cualquier secuencia de caracteres encabezada por “#” y terminada en “\n”, “Fin de línea” o “Fin de fichero”.
- Casts: Cualquiera de “int”, “hex”, “bin”, “oct”, “float”, “str”, “chr”
- DefFunc: Cualquiera de “pow”, “round”, “len", “abs”, “random”, “range”, “max”, “min”, “toLowerCase”, “toUpperCase”, “exp”, “timestamp”, “split”, “replace”, “ceil”, “notify”, “indexOf”, “lastIndexOf”, “charAt”, “log”, “sqrt”, “time”, "ctime”, “search”.
- UtilFunc: Cualquiera de “is_int”, “is_hex”, “is_bin”, “is_oct”, “is_float”, “is_str”.
- OtraCosa: Cualquier otra combinación de caracteres.

A continuación se definen los diferentes token directamente mediante una definición o en base a estas categorías.

Blacklisted	'black'	“(” o “)”	'par'
Binary	'bin'	“[” o “]”	'corch'
Octal	'oct'	“{” o “}”	'llave'
Hexadecimal	'hex'	“;”	'pyc'
Int	'int'	“:”	'pyp'
Real	'real'	“,”	'coma'
Reservada	'pr'	“=”	'asig'
Bloque	'blk'	Directiva	'direct'
DefFunc	'df'	Comentario	IGNORAR.
UtilFunc	'df'	“” o “Fin de fichero”	'EoF'
Boolean	'bool'	Secuencia de “\t”	IGNORAR.
OperadorLog	'oL'	Secuencia de “ ”	IGNORAR.
OperadorLogBit	'oLB'	“\n” o “Fin de línea”	'EoL'
OperadorDesBit	'oDB'	Casts	'df'
Cadena	'str'	Nombre	'id'
OperadorArit1	'oA1'	Path	'path'
OperadorArit2	'oA2'	OtraCosa	'ERR'
OperadorRel	'oR'		

Las combinaciones marcadas como IGNORAR, secuencias de espacios, tabuladores y comentarios, serán ignoradas por el analizador léxico procediendo inmediatamente a capturar el siguiente token. De esta forma estas combinaciones son completamente ignoradas por el análisis sintáctico evitando tener que ser procesadas.

Cada token se compondrá del identificador indicado y la cadena de texto que lo ha generado para un análisis más extenso si lo requiere el analizador sintáctico. Se incluirá información relativa a la posición del token dentro del código para la generación de errores más específicos.

3.8.3 Gramática del sintáctico

Esta sección presenta la gramática utilizada en el análisis sintáctico del código GEMA-Script. Se exponen en primer lugar los símbolos terminales de la misma que corresponderán con los token que no han de generar más producciones:

path	(Variables/propiedades/atributos)	&	(And binario)
id	(Nombre de variable local)	~	(Negación binaria)
bool	(true, false)	<	(Menor que)
int	(Entero: X)	>	(Mayor que)
real	(Real: X.Y)	<=	(Menor o igual)
hex	(Hexadecimal: 0xYYY)	>=	(Mayor o igual)
bin	(Binario: 0bXX.)	==	(Igual)
oct	(Octal: 0XXX)	!=	(Distinto)
str	(Cadena de texto: "Lorem ipsum")	=	(Asignación)
(	(Paréntesis abierto)	<<	(Desplazamiento binario a la izquierda)
)	(Paréntesis cerrado)	>>	(Desplazamiento binario a la derecha)
[	(Corchete abierto)	def	(declaración de funciones)
]	(Corchete cerrado)	if	(Estructura condicional "If")
{	(Llave abierta)	elsif	(Alternativa condicional, "Select")
}	(Llave cerrada)	else	(Alternativa general de la condicional)
+	(Suma)	for	(Estructura "For")
-	(Resta)	in	(Declaración del recorrido del "For")
*	(Producto)	while	(Estructura "While")
/	(División)	end	(Fin de bloque de sentencias)
%	(Módulo)	return	(Declaración del valor a devolver)
||	(Or)	static	(Declaración de variable estática)
&&	(And)	df	(int, hex, bin, oct, float, str, abs, random)
!	(Negación)	eol	(Fin de línea)
|	(Or binario)	eof	(Fin de archivo)
^	(Or Exclusivo binario)		

A continuación se exponen los símbolos no terminales de la gramática. Estos símbolos representarán las diferentes producciones de la gramática:

$:	Programa, Axioma de la gramática.	M:	Términos de ^
B:	Bloque de sentencias	N:	Términos de &
D:	Declaración de función	S:	Términos de <<, >>, >>>
G:	Declaración de parámetros	X:	Expresión aritmética
A:	Sentencia de asignación a "id"	T:	Términos de +,-
C:	Sentencia asignación a "path"	I:	Términos de *, /, %
F:	Sentencia "If"	Q:	Constante
J:	Sentencia "For"	R:	Vector de constantes
K:	Sentencia "While"	Z:	Direccionamiento de vector
V:	Parámetros de llamada a función	W:	Serie de expresiones aritméticas
E:	Expresión Lógica	P:	Serie de expresiones lógicas
O:	Términos de ||	U:	Serie de constantes
Y:	Términos de &&	H:	Serie de "id"
L:	Términos de |		

Finalmente se presentan las producciones de la gramática. Cada una de estas producciones representará el código condicional de las funciones del analizador sintáctico. Estas funciones vendrán definidas por los símbolos no terminales de la gramática, correspondiéndose una función por cada uno de ellos.

$	→	B eof	Y	→	X Y1
B	→	path C eol B	Y1	→	opr X
B	→	id A eol B	Y1	→	λ
B	→	return E eol B	S	→	X S1
B	→	if F B	S1	→	oDB X S1
B	→	def D B	S1	→	λ
B	→	static id = Q eol B	X	→	T X1
B	→	static id = R eol B	X1	→	+ T X1
B	→	while K B	X1	→	- T X1
B	→	for J B	X1	→	λ
B	→	end eol	T	→	I T1
C	→	= E eol B	T1	→	* I T1
R	→	[Q U]	T1	→	/ I T1
U	→	, Q U	T1	→	% I T1
U	→	λ	T1	→	λ
J	→	id in E eol B	I	→	- I
K	→	E eol B	I	→	! I
D	→	id G eol B	I	→	(E)
G	→	( )	I	→	[E P]
G	→	(id , H)	I	→	[ ]
H	→	, id H	I	→	path
H	→	λ	I	→	id
F	→	E eol B F1 F2	I	→	id Z
F1	→	elsif E eol B F1	I	→	id V
F1	→	λ	I	→	id ( )
F2	→	else B	I	→	Q
F2	→	λ	Z	→	[X W]
E	→	O E1	Z	→	λ
E1	→	|| O E1	W	→	, X W
E1	→	LAMB	W	→	λ
O	→	L 01	V	→	(E P)
O1	→	&& L O1	V	→	λ
O1	→	λ	P	→	, E P
L	→	M L1	P	→	λ
L1	→	| M L1	Q	→	true
L1	→	λ	Q	→	false
M	→	N M1	Q	→	int
M1	→	^ N M1	Q	→	real
M1	→	λ	Q	→	hex
N	→	Y N1	Q	→	oct
N1	→	& Y N1	Q	→	bin
N1	→	λ	Q	→	str

La gramática se diseña cumpliendo la condición LL1. Esto implica que las producciones derivan por la izquierda y que únicamente viendo el siguiente token es posible tomar la decisión sobre la siguiente producción a evaluar. Esto permite simplificar la implementación de las condiciones en el código de cada producción.

3.8.4 Implementación: Analizador recursivo predictivo

La implementación del analizador sintáctico sigue el modelo de analizador descendente recursivo predictivo. Para ello se apoya en una gramática LL1 que permita un análisis descendente sin retroceso con la única evaluación del siguiente token en la entrada.

Para ello se implementa una función por cada símbolo no terminal de la gramática. Esta función analizará los tokens a medida que van siendo reconocidos. Para determinar la expansión que debe realizarse utilizará el siguiente token proporcionado por el analizador léxico. Una vez determinada evaluará los símbolos terminales que vayan entrando y procederá a llamar a las funciones "no terminales" que correspondan a la producción realizada.

Un ejemplo de la implementación de la Producción T sería el siguiente:

T → I T1 T1 → * I T1 T1 → / I T1 T1 → % I T1 T1 → λ	*def procesa_T(self, estado):* *'''Procesa los términos de una suma o resta.'''* *expr = self.procesa_I(estado)* *if not expr:* *return expr* *if estado.token[0] == 'oA2':* *operador = estado.token[1]* *estado.next()* *expr1 = self.procesa_T(estado)* *if not expr1:* *return expr1* *expr = '{0} {1} {2}'.format(expr, operador, expr1)* *return expr*

Observamos que en primer lugar se realiza una llamada a la función "procesa_I" almacenando su resultado en la variable "expr". Esta llamada se debe al primer símbolo no terminal de la producción: T → I T1.

Si no ocurrió ningún error durante la ejecución de "procesa_I" evalúa el tipo de token actualmente apuntado por el analizador léxico. Almacena el operador y se llama a si misma recursivamente mientras no haya errores en la ejecución de "procesa_I" y el siguiente token sea siempre 'oA2', un operador aritmético de segundo nivel "*", "/" o "%".

Una vez terminada la cadena de operadores compone la expresión a devolver a la llamada anterior, retrocediendo recursivamente hasta finalizar la cadena y devolver la expresión completa a la función superior que inicio este análisis.

Dado que el código generado en la expresión será posteriormente analizado por el intérprete de Python no se hace una evaluación exhaustiva ni necesariamente ordenada, analizándose únicamente la estructura general de la expresión.

En el siguiente ejemplo podemos observar la implementación de "procesa_C" para la evaluación de una asignación a un "path": C → = E eol B

```
def procesa_C(self, estado):
    '''Procesa una sentencia de asignación a path.'''
    path = estado.token[1]
    estado.next()
    if estado.token[0] == 'asig':
        estado.next()
        expr = self.procesa_E(estado)
        if expr:
            if estado.token[0] == 'EoL':
                traduc = self.traduce_nodo_escritura(path, expr)
                self.add_refs(traduc, estado.refs)
                estado.emite('{0}{1}\n'.format(estado.level(), traduc))
                estado.next()
            else:
                estado.source = error(err_code, err_msg, err_data)
        else:
            estado.source = expr
    else:
        estado.source = error(err_code, err_msg, err_data)
```

En el código se observa como esta función procesa el path analizado por la función que la llame, "procesa_B", esto se debe a la necesidad de procesar conjuntamente la expresión y el path en la función "traduce_nodo_escritura" para generar el código de Python.

Una vez almaceno este valor evalúa que el siguiente token es el símbolo terminal de asignación. Una vez verificado avanza el puntero de lectura y llama a "procesa_E" la cual devolverá la expresión traducida de la expresión o un Error.

Si la expresión se evaluó correctamente se llama a la función de traducción de los paths, se añaden las referencias y se procede a emitir el código en Python ya generado. Tras avanzar el puntero de lectura la función finaliza su ejecución.

En caso de error en cualquiera de las llamadas o comprobaciones la función devolverá un Error que será propagado por la función superior hasta alcanzar la función principal del Parser y ser devuelto como resultado de la compilación.

Todo el proceso de compilación se apoyará en la clase Estado. Esta clase incluye el analizador léxico que va extrayendo los tokens del código fuente a medida que se le soliciten. Es también la responsable de mantener las tablas de símbolos de los identificadores declarados y el nivel de anidamiento de las mismas.

Para su implementación se utiliza una pila de diccionarios. Serán las propias funciones de producción las que soliciten descender o ascender en la misma en función de las declaraciones encontradas.

El objeto Status también es el encargado de almacenar el código intermedio Python generado así como su nivel de indentación. El nivel de indentación en Python define los bloques de sentencias de una estructura como se observa en los ejemplos expuestos.

Junto con el código generado almacenará las listas de referencias a los elementos de GEMA accedidos en los paths procesados por el analizador.

Dado que el código generado por el compilador será posteriormente analizado por el intérprete de Python se realizan pocas comprobaciones semánticas. Con respecto a los tipos, dado el tipado dinámico y la incertidumbre a la hora de conocer los tipos de las variables y propiedades accedidas únicamente se comprueba que las asignaciones se realicen sobre identificadores de variables o paths de GEMA.

Las expresiones aritméticas no se evaluarán necesariamente en el correcto orden de evaluación requerido por los operadores. Esto simplifica su evaluación al hacerse directamente en el orden introducido.

Estas expresiones serán traducidas a Python exactamente como se introdujeron en el código GEMA-Script. Será el intérprete Python el encargado de realizar la correcta evaluación de las mismas a la hora de ejecutarlas.

Para la implementación del analizador léxico se ha utilizado la librería de análisis léxico de Python Plex [32]. Esta librería permite una definición de la gramática léxica en la misma estructura presentada anteriormente.

Algunos ejemplos de declaración para algunas de las definiciones serían los siguientes:

Dígito: Cualquier valor entre 1 y 9 ambos incluidos.

self.digit = Range("09")

Letra: Cualquier carácter alfabético.

self.letra = NoCase(Range("AZ"))

Cadena: Cualquier secuencia de caracteres encabezada por “ o r” y terminada por “.

self.cadena = Str('"', 'r"') + Rep(AnyBut('"') + Rep(Str('\\"'))) + Str('"')

Reservada: Cualquiera de “def”, “if”, “for”, “return”, “static”, “while”

self.reservada = NoCase(Str("def", "if", "for", "return", "static", "while"))

Comentario: Cualquier secuencia de caracteres encabezada por “#” y terminada en “\n”, “Fin de línea” o “Fin de fichero”.

self.comentario = NoCase(Str("#")) + Rep(AnyBut("\n")) + Alt(Str("\n"),Eol,Eof)

Para la construcción del analizador se realiza una asignación de tokens similar a la del diseño, a continuación se muestra una declaración reducida del mismo:

```
self.lex = Lexicon([
    (self.blacklisted,      'black'),
    (self.binary,           'bin'),
    (Any('[]'),             'corch'),
    (Str(';'),              'pyc'),
    (Str("="),              'asig'),
    (Str('')|Eof,           'EoF'),
    (Rep1(Any("\t")),       IGNORE),
    (Any("\n")|Eol,         'EoL'),
    (AnyChar,               'ERR')])
```

El código de Plex “IGNORE” indicará al analizador que debe ignorar el token.

Como se puede observar la sintaxis de declaración de gramática apara el léxico proporcionada por Plex es altamente intuitiva y fácil de corresponder con la diseñada. Esto la hace especialmente legible sin necesidad de un alto conocimiento de expresiones regulares y permitirá modificar fácilmente la gramática léxica del lenguaje de ser necesario.

Los tokens generados por el analizador léxico de Plex son procesados por la función “procesa”. Esta función revisa el formato de los tokens generados, reduciendo a minúsculas todas las palabras reservadas y añadiendo la información de posición del token dentro del código. Haciendo coincidir con el formato de token utilizado por el analizador sintáctico.

El formato utilizado consiste en una terna que contendrá la información necesitada por el analizador sintáctico.

(“oA2”, “”, (28, 8))*

Esta terna contendrá en primer lugar el código o tipo del token, en segundo lugar la cadena de texto que lo ha generado y en tercer lugar la información de posición del token en el código. Esta posición se indicará por la fila y columna en la que se encuentra dentro del texto del código GEMA-Script.

3.8.5 Traducción Python

En esta sección se exponen diferentes estrategias de traducción entre GEMA-Script y Python. Se dará una visión general de estas contrastando las estructuras del lenguaje de scripting con las implementadas en Python y se ahondará en los detalles de implementación en Python de algunas de las características de GEMA-Script.

El diseño de GEMA-Script, inicialmente un subconjunto de Java Script, se ha ido adaptando tanto a las necesidades de uso del lenguaje como al hecho de que este será posteriormente traducido a Python para ser interpretado. Se ha tratado de simplificar el proceso de traducción al tiempo que el lenguaje GEMA-Script sin que por ello perdiese capacidad de expresión.

Operadores aritméticos:

Los operadores aritméticos son los mismos en GEMA-Script, que los hereda de Java Script, que en Python. Sin embargo los operadores lógicos de Java Script, heredados de C, "&&" y "||" son traducidos a los operadores de Python "and" y "or" respectivamente.

Como se ha explicado anteriormente las expresiones son evaluadas en el orden introducido en lugar del adecuado matemáticamente debido a que una vez traducidas serán evaluadas de nuevo por el intérprete de Python. De esta forma la traducción de una expresión compuesta quedaría de la siguiente forma:

GEMA-Script: *g = 1 * 2 + (3 % 5) / ((8 – 9) * 6) <= 7 / 6 * (pow(5,5) -4) || (false && true)*
Python: *g = 1 * 2 + (3 % 5) / ((8 - 9) * 6) <= 7 / 6 * (pow(5,5) - 4) or (False and True)*

Se mantiene el orden especificado por el programador así como los paréntesis declarados. Queda a cargo del intérprete su evaluación matemática durante la ejecución.

Direccionamiento de vectores:

Para la traducción del direccionamiento de los vectores multidimensionales se ha de traducir la sintaxis compacta de GEMA-Script vector[1,3,2] a la sintaxis de Python vector[1][3][2]. Esto se debe a que en Python el vector multidimensional no existe.

En Python, al igual que en GEMA-Script, los vectores multidimensionales se construyen creando vectores de vectores. Una matriz tridimensional se definiría, de la siguiente manera:

Matrix2x2x2 = [[[0, 0], [0, 0]], [[0, 0], [0, 0]]]

GEMA-Script simplifica su direccionamiento permitiendo una sintaxis compacta. En Python cada direccionamiento dentro de un vector nos devuelve el elemento. Este elemento ha de ser direccionado hasta alcanzar el nivel requerido.

Esto permite en GEMA-Script direccionar solamente una fila, o una matriz bidimensional en el caso anterior, omitiendo el último valor:

GEMA-Script: *g = Matrix2x2x2[1,0]* → *[0,0]*
Python: *g = Matrix2x2x2[1][0]* → *[0,0]*

Implementación accesos:

Para los accesos a los elementos monitorizables de GEMA se diseña una sintaxis específica. Esta sintaxis, denominada "path", consiste en una serie de elementos direccionados dentro de la jerarquía del Layout separados por punto y coma:

*Nombre_Nodo;[Alias_Hijo;]*Nombre_Variable/Componente*

Para la traducción a Python de estos accesos se distingue entre su uso de lectura o escritura. De esta forma los elementos del path son recorridos uno a uno generando la cadena de direccionamiento en Python.

En los siguientes ejemplos podemos observar la traducción de un path tanto en modo de lectura, como en modo de escritura.

Lectura:

GEMA-Script: *Var1 = Nodo;ChA;Status*
Python: *Var1 = SL.Lay.acc_elto("Nodo").acc_subelto("ChA").get_componente("Status")*

Escritura:

GEMA-Script: *Nodo;ChA;Status = "False"*
Python: *SL.Lay.acc_elto("Nodo").acc_subelto("ChA").set_componente("Status","False")*

Lectura del "Tipo":

GEMA-Script: *Var1 = Nodo;ChA;Status;__tipo__*
Python: *Var1 = SL.Lay.acc_elto("Nodo").acc_subelto("ChA").get_tipo_componente("Status")*

En este punto existe un acoplamiento entre la implementación del Parser con el interfaz de métodos ofrecido por los elementos de GEMA. Cualquier cambio en el interfaz de estos elementos implicaría modificar las funciones de traducción y cálculo de referencias del Parser.

Implementación funciones:

GEMA-Script permite declaración de funciones tanto en el cuerpo principal de la propiedad como dentro de otras funciones.

```
var = 10
def operar(var)
def doble(var)
var = var * 2
return var
end
var = doble(var) + 1
return var
end
Res = operar(var)
return Res
end
```

En este código se implementa una función, "operar", dentro de la propiedad. A su vez, dentro de esta función, se implementa la función "doble".

Esta segunda función solamente será visible desde el cuerpo de la primera y no desde el cuerpo principal de la propiedad.

En GEMA-Script la indentación es totalmente opcional, siendo los delimitadores de bloque "end" los que determinan el final de cada bloque. Sin embargo en Python la indentación es en sí misma la declaración de bloques.

Esto es tenido en cuenta incrementando el nivel de indentación en el estado del Parser por cada palabra reservada que inicie un bloque y reduciéndolo con cada sentencia "end".

El código anterior traducido a Python quedaría de la siguiente manera:

```
var = 10
def operar(var):
    def doble(var):
        var = var * 2
        return var
    var = doble(var) + 1
    return var
Res = operar(var)
return Res
```

Como se puede observar los delimitadores “end” han sido reemplazados por los niveles de indentación adecuados a cada definición y se ha añadido el carácter “:” que inicia los bloques en Python. El resto del código permanece inalterado debido a que se utilizó la misma sintaxis de declaración de funciones en GEMA-Script que la existente en Python.

Durante el proceso de la declaración de los parámetros de la función se aumenta el “scope” de la tabla de símbolos, quedando los parámetros definidos únicamente dentro del cuerpo de la función.

De este modo la variable “var” a nivel del cuerpo de la propiedad es totalmente independiente de la definida dentro del cuerpo de cada una de las funciones declaradas. La ejecución del código mostrado devolvería como resultado el valor 21.

Condicionales y control de bucles:

Las sentencias condicionales y las de control de bucle: “if”, “elsif”, “else”, “for” y “while” se apoyan en la misma estrategia que la declaración de funciones para el control de la indentación de los bloques.

Al igual que en el caso anterior la sintaxis se ha asimilado de Python directamente para facilitar el proceso de traducción. En el siguiente ejemplo se observa un uso de estas estructuras anidadas:

GEMA-Script:

```
    stop = 0
    while stop == 0
        for f in range(0,9)
            if f == 1
                q = 0
            elsif f % 2 == 0
                q = "par"
            elsif f == 7
                stop = 1
            else
                q = 1
            end
        end
    end
end
```

Este código no tiene ninguna funcionalidad específica más allá de representar el uso de las estructuras citadas. Una vez traducido a Python por el Parser quedaría de la siguiente manera:

PYTHON:

```
stop = 0
while stop == 0:
    for f in range(0,9):
        if f == 1:
        q = 0
    elif f % 2 == 0:
        q = "par"
    elif f == 8:
        stop = 1
    else:
        q = 1
```

Se observa que las sentencias “end” se sustituyen por las indentaciones correspondientes y se añade el carácter “:” de inicio de bloque. También se puede comprobar la sustitución de la palabra reservada “elsif” por la palabra “elif” utilizada en Python.

Implementación variables estáticas:

Para la implementación de las variables estáticas se aprovecha una característica de los valores por defecto en los parámetros de función de Python. En Python cuando una función se declara con valores por defecto para alguno de sus parámetros este solamente es instanciado y creado la primera vez que se ejecuta.

```
Def funcion(var=[0]):
    ...
```

Con esta declaración, la primera vez que se llame a la función sin proporcionar parámetros se instanciara el objeto [0,0]. Cada ejecución posterior de la función sin parámetros acudirá a este objeto como valor por defecto.

Si durante la ejecución de la función este objeto es modificado la siguiente llamada lo recibirá modificado como valor por defecto:

```
def funcion(var=[0]):
        var[0] = var[0] + 1
        print var
```

```
>>> funcion()
[1]
>>> funcion()
[2]
>>> funcion()
[3]
>>> funcion()
[4]
```

De esta forma conseguimos un comportamiento de variable estática que recuerda su estado de ejecución anterior. Para aprovechar esta característica de Python cada declaración de variable estática en el código de GEMA-Script es convertida en una entrada de diccionario.

Este diccionario será declarado como valor por defecto de la variable "static" en la cabecera de la función de la propiedad. Por esta razón "static" es una de las palabras prohibidas dentro del código GEMA-Script.

La siguiente declaración y uso de variables estáticas en un código de propiedad:

```
static var      = 10
static memo     = [0,0,0,0,0,0,0,0,0,0]
static actual   = 0
```

Generaría la siguiente cabecera de función y código para la propiedad.

```
def code(self, mensaje=None, static={'var': 10, 'memo': [0,0,0,0,0,0,0,0,0,0], 'actual': 0}):
    static["actual"] = static["actual"] + 1
    static["memo"][5] = static["var"] * 3
```

Como se puede observar, las referencias a la variable estática en GEMA-Script se realizan normalmente. En Python estás son completadas añadiendo el acceso a la entrada correspondiente del diccionario de variables estáticas.

Estas entradas conservarán su valor e ejecución en ejecución pues nunca será declarado un valor para este parámetro durante la ejecución de una propiedad en GEMA.

Implementación de la cabecera, imports y declaraciones por defecto:

Como ya se vio en el ejemplo del apartado anterior la cabecera de la función generada contiene más información.

```
def code(self, mensaje=None, static={'var': 10, 'actual': 0}):
```

En primer lugar observamos que la función declarada se denomina "code". Este nombre es el esperado por las propiedades para las declaraciones de su función de update y será el nombre que intenten ejecutar una vez habilitada.

Para habilitar la propiedad se ejecutará una vez el código generado por el Parser. Este código consiste en una declaración de función que, una vez ejecutada, creará o modificará el atributo code de la propiedad.

El parámetro mensaje, con valor por defecto "None", es la variable en la que se almacenará el mensaje que activó la propiedad en caso de tratarse de una propiedad asíncrona. En caso de ser una propiedad síncrona este parámetro no será instanciado en su llamada obteniendo como valor "None" para la ejecución.

Esta variable es accesible durante en el código de la propiedad tanto de lectura como de escritura. Sin embargo su escritura no provocará más efecto que la pérdida del mensaje original si lo hubiese y se recomienda encarecidamente no utilizar esta variable para almacenar otra información.

Tras la cabecera de la función podremos encontrar una serie de imports:

```
import SingleLay as SL
import SingleGemmer as SG
from GSutils import is_int
...
```

Estos imports permiten al código de la propiedad accede a los diferentes elementos necesarios de GEMA.

SingleLay permitirá el acceso a los elementos del Layout mediante los paths traducidos. Se utilizará el Alias "SL" de forma que sea fácilmente sustituible el módulo de Layout sin necesidad de cambiar el código del Parser.

SingleGemmer permite el acceso al sistema de notificaciones GEMA, de esta forma una propiedad podrá lanzar nuevas notificaciones. El uso del alias "SG" tiene la misma justificación que en el caso anterior.

GSUtils incluye todas las funciones del módulo Utils definidas anteriormente y su importación específica permite su uso directamente desde el código de la propiedad sin necesidad de especificar el módulo.

Finalmente, tras estos imports y justo antes del código traducido podemos encontrar la declaración de la función de notificación de mensajes:

```
def notify(msg, lvl):
    return SG.Listener.notify(msg, lvl, self.name)
```

Esta función permitirá crear notificaciones de mensajes GEMA desde el código de las propiedades sin necesidad de que el programador de GEMA-Script conozca el funcionamiento interno del módulo. La función incorporará automáticamente al mensaje su propio nombre. Este será utilizado, además de notificar a los subscriptores del remitente de la notificación, para evitar que una propiedad pueda reactivarse a sí misma mediante un mensaje.

Finalmente, el código completo de una propiedad quedaría de la siguiente manera:

GEMA-Script:

```
#%TYPE:Test
static var      = 10
static memo     = [0,0,0,0,0,0,0,0,0,0]
static actual   = 0
# Incremento circular de tamaño 'var'
def inc(i,var)
    i = i + 1
    if i == var
        i = 0
    end
    return i
end
memo[actual] = timestamp()
actual = inc(actual,var)
def ordena3(V,A,var)
    return [V[(A-1)%var],V[(A-2)%var],V[(A-3)%var]]
end
return ordena3(memo,actual,var)
end
```

Este código implementa una propiedad que almacena el "timestamp" de su ejecución en un buffer circular en el vector estático "memo". La función declarada "inc" controla el puntero de escritura, "actual", en el buffer, rotándolo al alcanzar la posición "var" dentro del buffer.

Posteriormente devuelve la lista de las últimas 3 ejecuciones por medio de la función "ordena3" que lo extrae del buffer en función del valor del puntero "actual". Una vez compilado y traducido por el Parser el código de la propiedad queda de la siguiente manera:

PYTHON:

```
def code(self, mensaje=None, static={'var': 10, 'memo': [0,0,0,0,0,0,0,0,0,0], 'actual': 0}):
    import SingleLay as SL
    import SingleGemmer as SG
    from GSutils import is_int
    from GSutils import is_hex
    from GSutils import is_bin
    ...
    def notify(msg, lvl):
        return SG.Listener.notify(msg, lvl, self.name)
    def inc(i,var):
        i = i + 1
        if i == var:
            i = 0
            return i
    static["memo"][static["actual"]] = timestamp()
    static["actual"] = inc(static["actual"],static["var"])
    def ordena3(V,A,var):
        return [V[(A - 1) % var],V[(A - 2) % var],V[(A - 3) % var]]
    return ordena3(static["memo"],static["actual"],static["var"])
```

En él se puede observar la cabecera completa con la declaración de la variable "mensaje" y del diccionario de variables estáticas.

Tras la cabecera encontramos los imports descritos anteriormente, (se omite la lista completa de imports específicos de GSUtils) y la definición de la función "notify" para los mensajes GEMA.

Finalmente encontramos el código traducido de la propiedad implementada anteriormente en GEMA-Script.

Junto con este código se entregará, a la función que solicitó el parse, las listas de referencias, vacías en este caso, y el tipo definido para la propiedad "Test" mediante la directiva implementada: *"#%TYPE:Test"*

Esta información será utilizada por el API de GEMA para editar la propiedad correspondiente. Una vez editada se ejecutará el código generado quedando redefinido el atributo ejecutable (método) "code" permitiendo su ejecución durante el proceso de update de las propiedades. Cualquier error generado durante la ejecución de la propiedad, será capturado por GEMA. Una vez capturado el error bloqueará la propiedad y lo notificará. Será responsabilidad del administrador revisar y corregir el código para su ejecución.

3.9 Vista global del diseño

En el siguiente diagrama se puede observar una vista global del diagrama de clases que conforma GEMA. En él se representan todas las diferentes relaciones entre las clases y las diferentes capas a las que pertenece cada una. Cada clase incluye la enumeración de sus atributos y métodos.

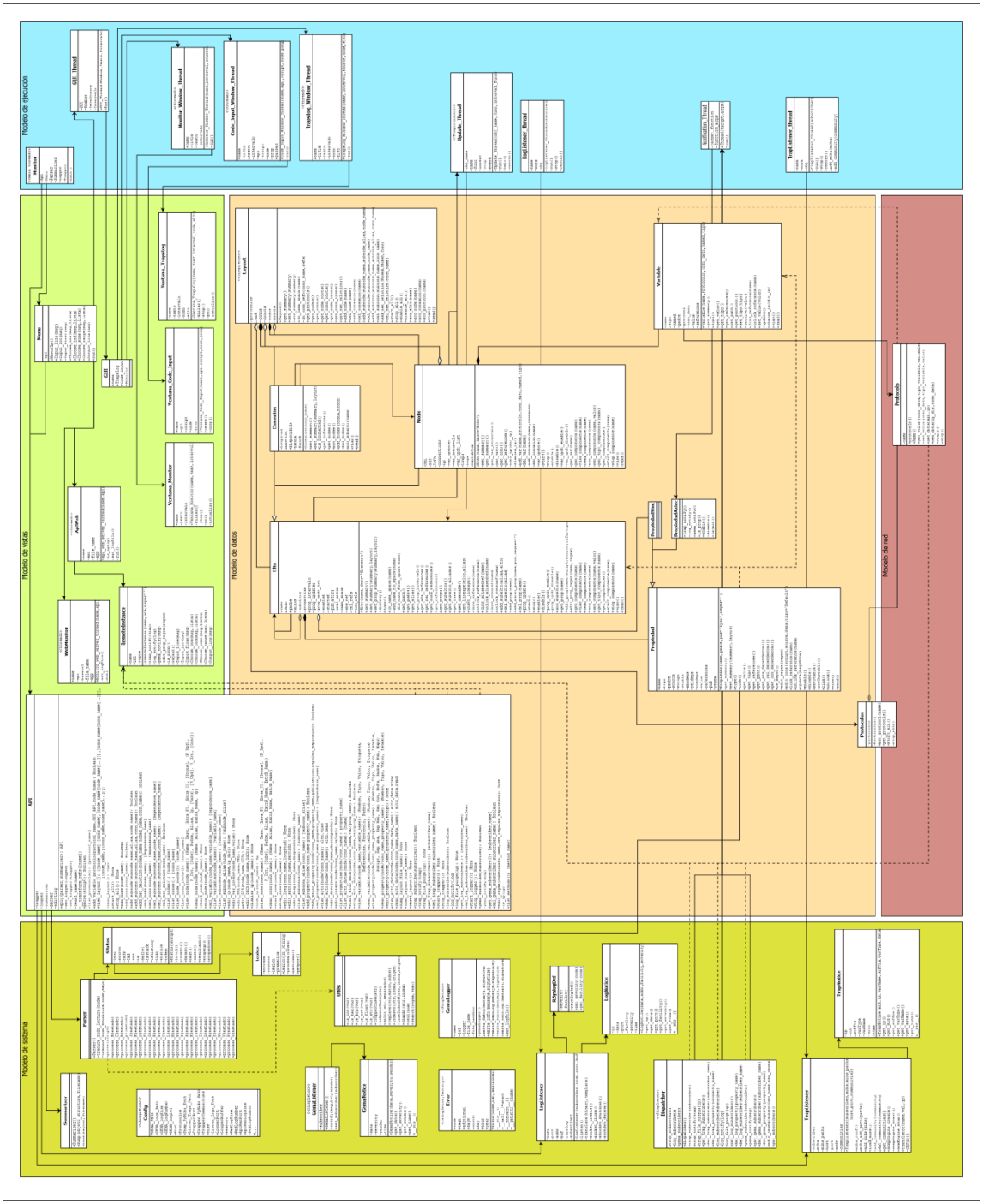


Figura 3.62. Diagrama completo de clases de GEMA.

Con este diagrama finaliza la descripción del diseño de aplicación realizado e implementado. En el siguiente capítulo se introduce la primera versión completa de GEMA ya desarrollada y probada, GEMAv1, que ha sido entregada a Fibernet.

Capítulo 4

GEMA

Índice

En esta capitulo se presenta la aplicación GEMA implementada. Se realiza un recorrido por los diferentes menús, ventanas y controles de los que dispone desde el punto de vista del usuario.

Todos los elementos descritos se han implementado como módulos independientes y realizan su labor a través del API. De esta forma es posible añadir nuevos o sustituir cualquiera de ellos por una versión más avanzada o completa.

4.1. Sistema de menús en consola

El sistema de menús de consola permite al administrador gestionar todos los elementos de GEMA. Desde estos menús podrá establecer el diseño de una red, gestionar la actividad de diferentes módulos y salvar o cargar diseños previos.

Estos menús se estructuran en forma arborescente, descomponiéndose en diferentes submenús según las diferentes categorías de operaciones a realizar. Esto permite organizar las diferentes funcionalidades según categorías.

Nada más lanzar la ejecución de la aplicación se presenta el menú principal de GEMA. Este menú incluye el acceso a la consola interactiva de Python. Durante la ejecución de esta consola el sistema de menús quedará bloqueado.

Menú principal:

```
1: Diseño          → Menú de diseño.
2: Complementos    → Menú de control de los complementos.
0: Salir           → Finaliza, de forma ordenada, la ejecución de la aplicación.
ts:                → Lanza la consola interactiva de Python.

>> _
```

Cada uno de estos submenús ofrece opciones directas a las funcionalidades o nuevos menús que especifican las opciones disponibles para su categoría.

En el menú de complementos encontramos las diferentes opciones de activación y uso de los complementos.

Menú de complementos:

1: Lanzar monitor	→ Lanza la ventana de monitorización local.
2: Lanzar Web	→ Activa el servicio web de monitorización.
3: Lanzar Api Web	→ Activa el servicio Api Web.
4: Lanzar Logger	→ Inicia el servicio de captura de Logs.
5: Lanzar Trapper	→ Inicia el servicio de captura de Traps.
6: Parar Logger	→ Detiene el servicio de captura de Logs.
7: Parar Trapper	→ Detiene el servicio de captura de Traps.
8: Monitorizar Traps	→ Lanza una ventana de monitorización de Traps.
9: Monitorizar Log	→ Lanza una ventana de monitorización de Logs.
10:Purgar subscriptor a los trap	→ Elimina un subscriptor de Traps.
11:Purgar subscriptor a los log	→ Elimina un subscriptor de Logs.
12: Purgar subscriptor a Gema	→ Elimina un subscriptor de Mensajes.
13: Purgar fichero de traps	→ Vacía un fichero de historial de Traps.
14:Purgar fichero de logs	→ Vacía un fichero de historial de Logs.
0: Volver	→ Regresa al menú principal.

>> _

En el menú de diseño se encuentran las diferentes opciones disponibles para la definición de la red.

Menú de diseño:

1: Añadir nodo	→ Añade un nuevo nodo al layout.
2: Añadir conexión	→ Añade una nueva conexión al layout.
3: Añadir relación	→ Establece una relación de nodos mediante una conexión.
4: Eliminar nodo	→ Elimina un nodo del layout.
5: Eliminar conexión	→ Elimina una conexión del layout.
6: Eliminar relación	→ Elimina una relación establecida entre dos nodos.
7: Ver/Editar nodo	→ Menú de edición de Nodos.
8: Ver/Editar conexión	→ Menú de edición de conexiones.
9: Guardar/Cargar diseño	→ Menú para salvar o cargar diseños.
0: Volver	→ Regresa al menú principal.

>> _

Cada una de las opciones de estos menús lanzará una serie de consultas en pantalla que permitirá resolver la opción solicitada o mostrar el siguiente menú en la jerarquía. Por ejemplo: al añadir un nuevo nodo se nos consultará su nombre, al solicitar editar un nodo se mostrará la lista de los existentes para seleccionar el nodo a modificar.

Los menús de edición permiten cambiar los valores de los parámetros de los nodos así como añadir o eliminar variables y propiedades.

Menú de edición de nodo:

```
1:  Añadir Variables          → Inicia el proceso de adición de variables al nodo.
2:  Añadir Propiedad          → Crea una nueva propiedad en el nodo.
3:  Añadir Subnodos           → Establece los subnodos.
4:  Eliminar Variable         → Elimina una variable del nodo.
5:  Eliminar Propiedad        → Elimina una propiedad del nodo.
6:  Eliminar Subnodo          → Retira un nodo hijo.
7:  Editar IP                 → Define un nuevo valor para la IP del nodo.
8:  Editar Descripción        → Define un nuevo valor para la descripción del nodo.
9:  Editar Intervalo de propiedad→ Define un nuevo intervalo de actualización de propiedad.
10:Editar Intervalo de variable → Define un nuevo intervalo de actualización de variable.
11:Editar Propiedad           → Edita el código de una propiedad del nodo.
12:Editar Expresión regular   → Edita la expresión regular de una propiedad del nodo.
13:Arrancar Elemento          → Activa la actualización de un elemento del nodo.
14:Parar Elemento             → Desactiva la actualización de un elemento del nodo.
0:  Volver                    → Menú de diseño.

>> _
```

Menú de edición de conexión:

```
1:  Añadir Propiedad          → Crea una nueva propiedad en la conexión.
2:  Añadir Subconexiones      → Establece las subconexiones.
3:  Eliminar Propiedad        → Elimina una propiedad de la conexión.
4:  Eliminar Subconexión      → Retira una conexión hija.
5:  Editar Descripción        → Define un nuevo valor para la descripción de la conexión.
6:  Editar Intervalo          → Define un nuevo intervalo de actualización de propiedad.
7:  Editar Longitud           → Define un nuevo valor para la longitud de la conexión.
8:  Editar Sentido            → Define un nuevo valor para el sentido de la conexión.
9:  Editar Disponibilidad     → Define un nuevo valor a la disponibilidad de la conexión.
10:Editar Propiedad           → Edita el código de una propiedad de la conexión.
11:Editar Expresión regular   → Edita la expresión regular de una propiedad.
12:Arrancar Elemento          → Activa la actualización de un elemento de la conexión.
13:Parar Elemento             → Desactiva la actualización de un elemento de la conexión.
0:  Volver                    → Menú de diseño.

>> _
```

Desde el menú de carga y salvado accedemos a las funcionalidades de persistencia de los diseños de GEMA.

Menú de salvado y carga de diseños:

```
1:  Salvar Diseño             → Salva el diseño actual a un fichero en disco.
2:  Cargar Diseño             → Carga el diseño salvado en un fichero en disco.
0:  Volver                    → Menú de diseño.

>> _
```

4.2. Sistema de ventanas local

Desde el menú de complementos visto anteriormente se pueden lanzar las diferentes ventanas de monitorización local de GEMA. Estas ventanas permiten, de una forma simplificada, realizar un seguimiento de la monitorización en curso.

En primer lugar está la ventana del monitor. Esta ventana mostrará, en tiempo real, toda la jerarquía de elementos monitorizados en GEMA junto a sus valores:

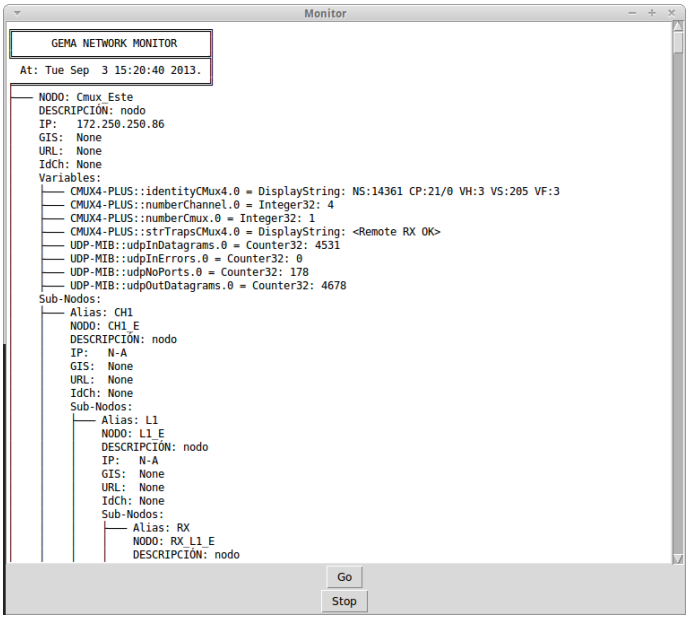
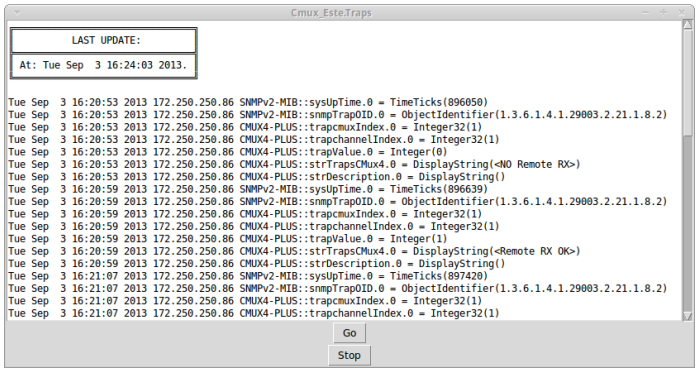


Figura 4.1. Ventana del monitor local.

Esta ventana dispone de un control para detener la actualización de la misma para una lectura cómoda, capturas de pantalla o toma de datos. Este control no detendrá la monitorización que seguirá ejecutándose en segundo plano. Tras pulsar de nuevo el contenido de la ventana se actualizará con los datos más recientes.

Las ventanas de monitorización de traps y logs, una vez seleccionado un nodo, mostrarán el historial de traps o logs recibidos desde el nodo seleccionado.



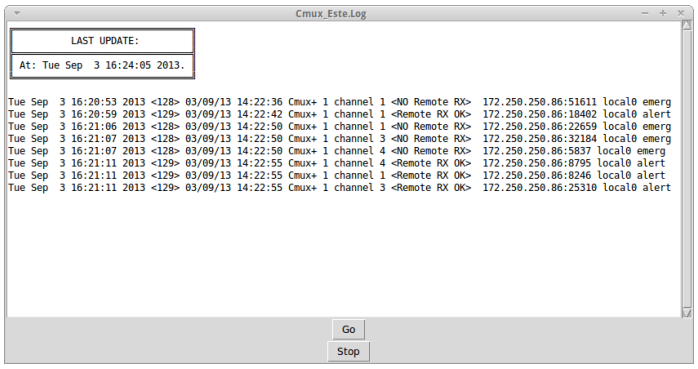


Figura 4.2. Ventanas de monitorización local de traps y logs.

Al igual que la ventana de monitorización esta ventana dispone de un botón de parada.

Es posible lanzar varias instancias de estas ventanas simultáneamente, tanto del mismo nodo como diferentes. Esto permite monitorizar al mismo tiempo las comunicaciones desde diferentes nodos y poder visualizar las notificaciones de forma simultánea.

Para la edición del código de propiedades se dispone de una ventana interactiva. Esta ventana se inicializará con el código actual de la propiedad.

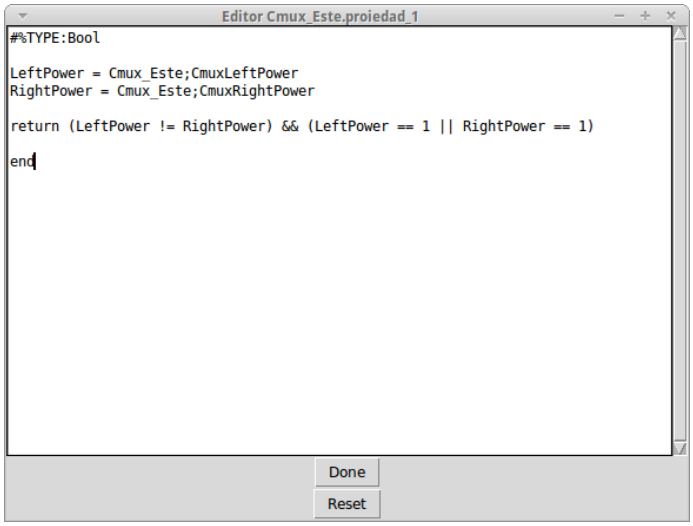


Figura 4.3. Ventana de edición de propiedades.

Desde esta ventana se podrá modificar el código GEMA-Script de la propiedad y solicitar su carga en el sistema. Tras su confirmación la ventana de edición se cerrará automáticamente devolviendo el control al menú de consola.

Si durante el proceso de análisis del código se produjese algún error este se mostraría en una ventana emergente indicándolo juntos con su posición en el código.

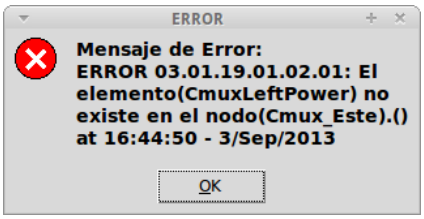


Figura 4.4. Ventana de notificación de error de compilación.

Tras cerrar esta ventana de error se devuelve el control al editor.

En cualquier momento es posible cerrar directamente la ventana de edición abortando así el proceso de modificación del código. Dispone igualmente de un botón que permite reiniciar el código de la propiedad a su estado inicial.

4.3. Servicio de monitorización web

El monitor web queda disponible una vez iniciado el servicio desde el menú de complementos. Este servicio creará una página web en la dirección y puerto configurados en el módulo de configuración.

La página web generada mostrará, en un formato similar al monitor local, la información de todos los elementos del layout. Esta ventana consta de accesos directos a los diferentes elementos permitiendo una vista más detallada de los mismos.

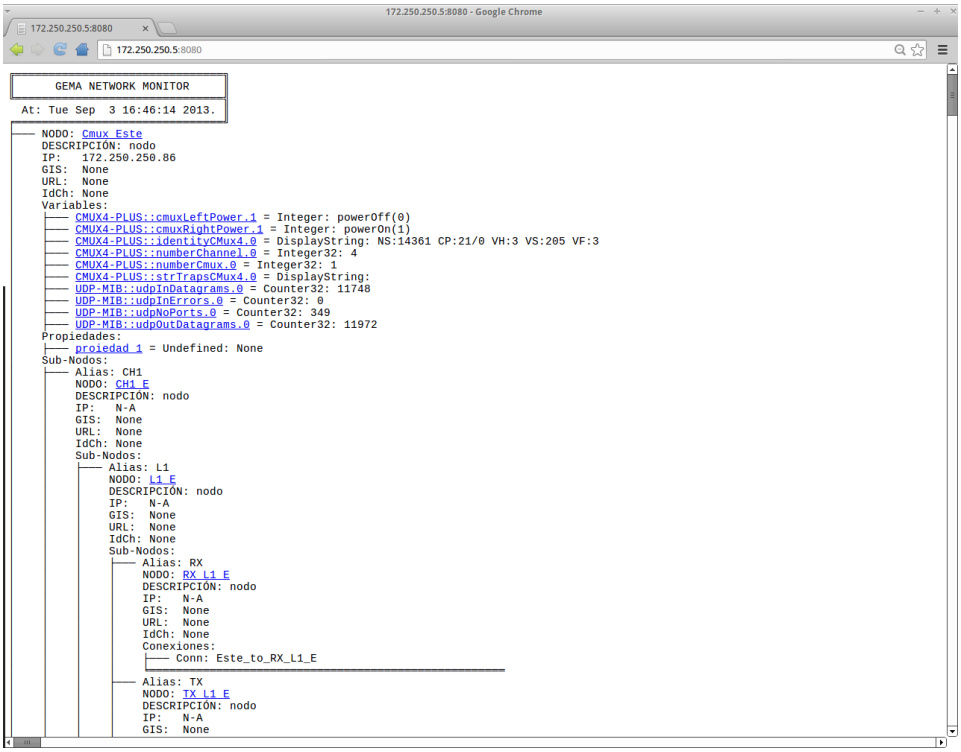


Figura 4.5. Página web del monitor.

Cada elemento monitorizado de GEMA podrá ser inspeccionado en mayor detalle accediendo por su acceso tanto desde la página principal como desde cualquiera anterior en su jerarquía en la que aparezca representado.

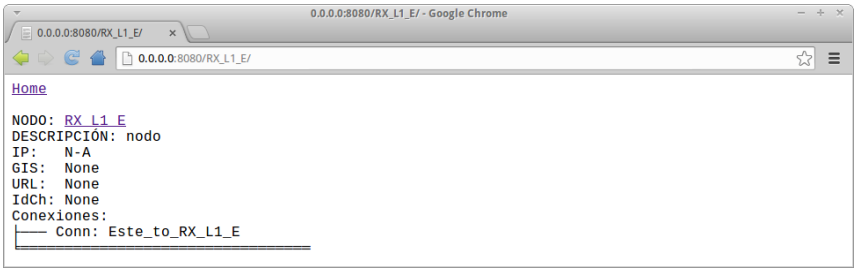


Figura 4.6. Página web del detalle de un nodo.

De esta forma es posible navegar en la estructura del layout monitorizado, accediendo a los diferentes elementos y su información detallada en tiempo real.

Todas las páginas generadas están programadas para recargar su contenido de forma automática, de esta forma el propio navegador se convierte en un monitor en tiempo real sin necesidad de ninguna configuración. Para detener la actualización se utilizarán los controles propios de cada navegador a tal efecto. Toda la programación de las páginas web está realizada exclusivamente en HTML, de esta forma no es necesario la disponibilidad de interpretes adicionales tales como podrían ser Java-Script o Flash.

4.4. Servicio del Api-Web

El servidor del Api-Web, activado desde el menú de complementos, crea un servicio web en la dirección y puertos especificados en el módulo de configuración. Este servidor atenderá peticiones GET y POST para los diferentes servicios disponibles en el Api.

Las aplicaciones cliente que deseen conectar con GEMA deberán hacerlo a través de este interfaz. Para ello deberán implementar las llamadas GET y POST requeridas contra el servicio del Api-Web de GEMA.

Adicionalmente, en la misma dirección del servicio, el servidor despliega una página web que resume los comandos disponibles.

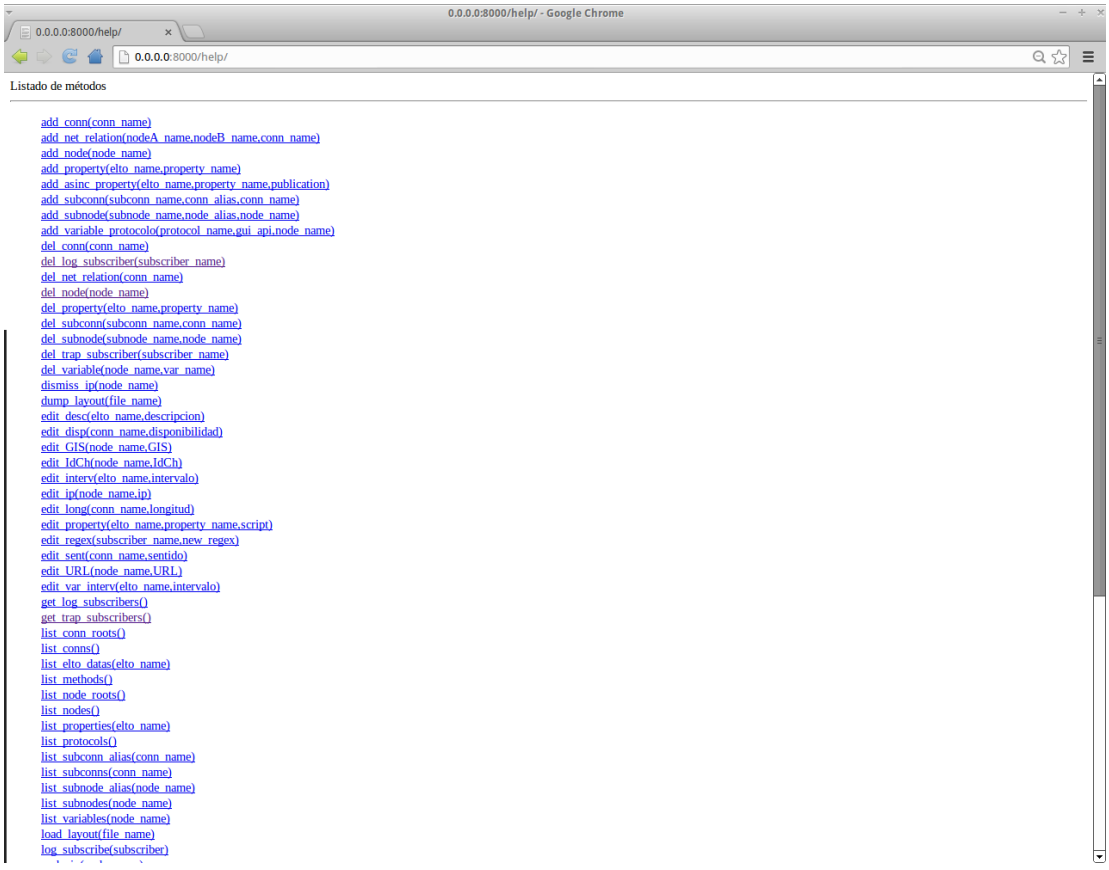


Figura 4.7. Página web del listado de métodos.

Cada entrada en el listado es un enlace a una página que describe en mayor detalle el método seleccionado.

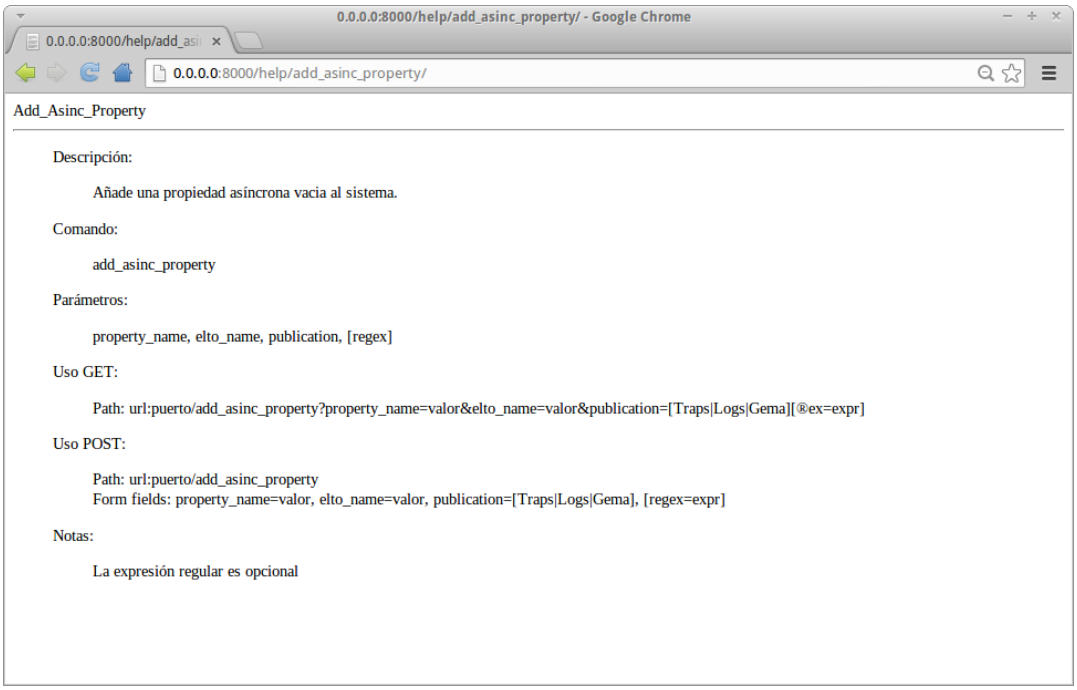


Figura 4.8. Página web del detalle de un método.

En esta descripción podemos encontrar una pequeña indicación de la funcionalidad del método, los parámetros que requiere, las diferentes líneas de comando GET y/o POST que utiliza y las notas adicionales convenientes.

Capítulo 5

Caso de uso

Índice

En esta sección se presenta un caso de uso completo. Comenzando por la instalación y despliegue de la aplicación en una máquina hasta diferentes casos de monitorización. Para ello se presentará un diseño sencillo de red a monitorizar y se irán comentando los diferentes pasos para su diseño en GEMA.

Se realizarán algunos ejemplos de programación de propiedades sobre las variables monitorizadas y, finalmente, se mostrarán diferentes alternativas de monitorización de la red.

5.1. Instalación y despliegue de GEMA

GEMA, al ser una aplicación Python, únicamente requiere para su instalación disponer de un intérprete de Python 2.7, de serie en casi todas las distribuciones Linux.

Se dispone de un script de instalación, install.sh, que descargará la aplicación desde un repositorio, construirá la estructura de directorios y permisos necesarios, descargará los paquetes de librerías necesarios y realizará todo el proceso de instalación.

Está disponible también un script, update.sh, que permitirá descargar una nueva versión desde el repositorio actualizando así la versión actual de GEMA en el sistema.

Para la ejecución de GEMA se deberá iniciar sesión con el usuario que ejecutó el proceso de instalación. Desde un terminal de consola de este usuario ejecutaremos el mandato "*gema-console*". Tras la ejecución de este mandato aparecerá en consola el menú principal de la aplicación.

No se requiere ninguna instalación adicional en equipos remotos para acceder a las opciones de monitorización remota. Únicamente deberán de disponer de una aplicación que conecte con el Api-Web de GEMA y la información de conexión al servidor.

5.2. Alta de un diseño de Layout

En primer lugar definiremos la red a monitorizar. Esta red consta de dos CMUX4+. Antes de entrar en los detalles de la red, se hace una breve explicación de que es un CMUX4+ y de que bloques consta.

El equipo CMUX4+ multiplexa las señales de los distintos canales que estén activos. Para ello posee un multiplexor (**MUX**). Por otro lado, está formado por 4 canales (**Ch**) que constan de transceivers (SFPs con un Tx y un Rx) uno hacia el lado del cliente, **Loc** (hacia la nube) y otro **Rem** (hacia el multiplexor). Por otro lado, el llevará una o dos fuentes de alimentación (**FA**) conectadas de distintos tipos y finalmente una unidad de ventilación (**UV**).

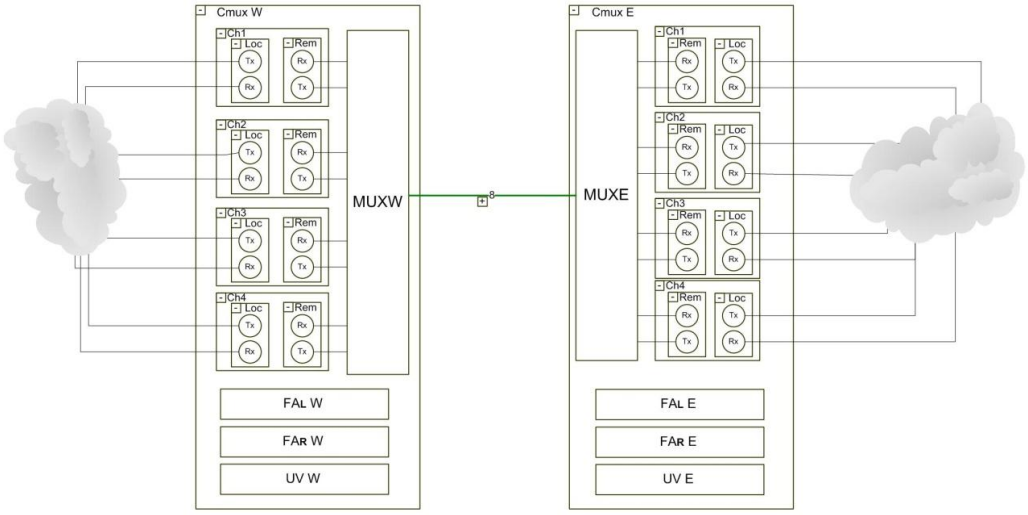


Figura 5.1. Diagrama de la red. 2 CMUX4+ conectados entre sí.

Hay dos clientes oeste (**W**) y este (**E**), que envían sus señales a través de los SFP **Loc**. Los SFPs **Rem** son reciben y envían las señales al otro CMUX mediante el multiplexor (**MUX**).

A continuación se realiza un diseño de representación en árbol del layout que será implementado en GEMA.

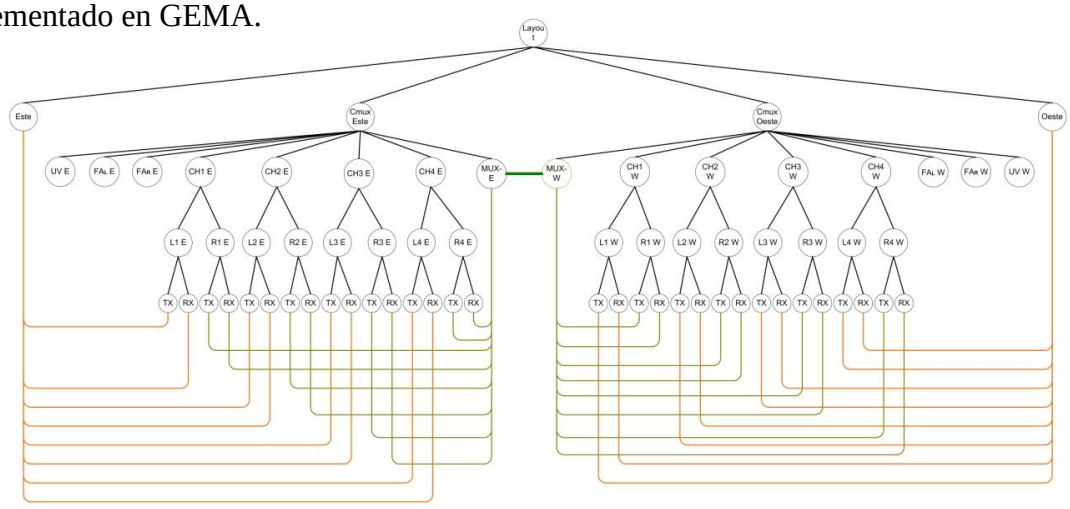


Figura 5.2. Representación en árbol de la red.

5.2.1. Alta de nodos

A continuación se dan de alta todos los nodos en GEMA, 68 en total. Para poder observar la evolución del diseño crearemos una ventana de monitorización desde el menú de complementos.

Para dar de alta los nodos accederemos al menú de diseño por consola. Para cada nodo utilizaremos la primera opción de este menú. Esta opción nos pedirá el nombre de cada nodo y, una vez verificado que este es correcto y único, se dará de alta en el sistema.

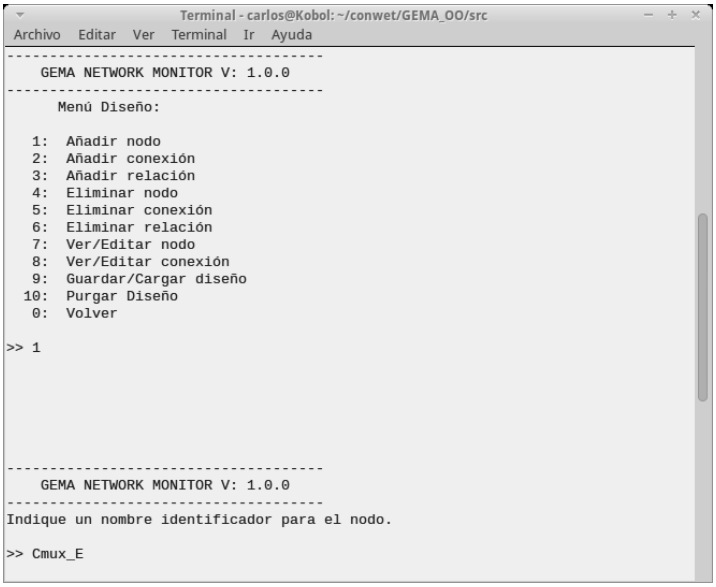


Figura 5.3. Alta de nodos en GEMA.

Una vez dados de alta todos los nodos se procede a establecer la jerarquía especificada en la figura 5.2. Para ello accederemos al menú de edición de cada nodo. En este menú la tercera opción nos mostrará una lista de nodos disponibles.

De esta lista seleccionaremos los hijos del nodo actual. Para cada nodo hijo el sistema nos solicitará un alias. Este alias será utilizado para referenciar los nodos desde su padre de una forma genérica en la implementación de propiedades. Los alias deberán ser únicos dentro del espacio de nombres de cada nodo.

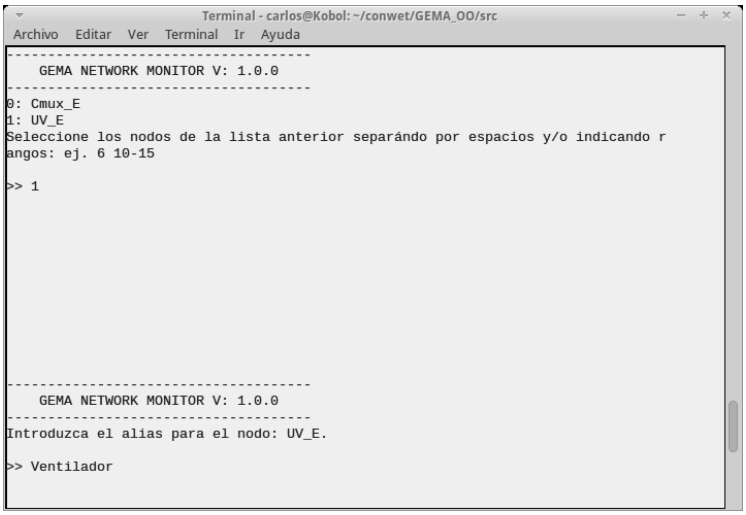


Figura 5.4. Establecimiento de nodos hijo en GEMA.

Tras realizar este proceso para cada nodo con hijos queda completada el alta de nodos y su jerarquía. Podemos comprobar en todo momento el resultado actual en la ventana de monitorización que creamos al principio.

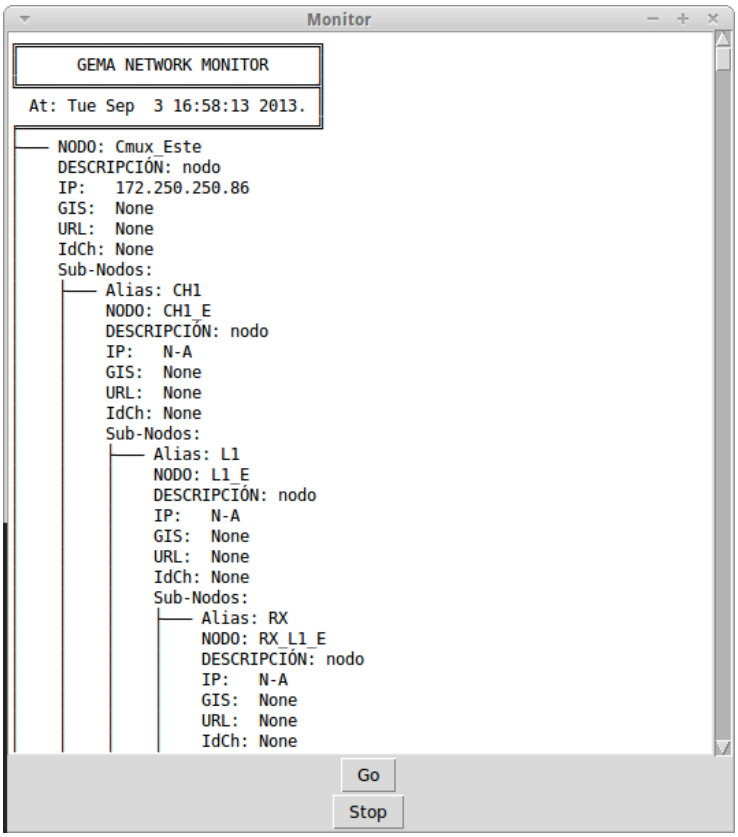


Figura 5.5. Resultado del alta de la jerarquía de nodos en GEMA.

5.2.2. Alta de conexiones

El proceso de alta de conexiones es idéntico al de los nodos. Para ello utilizaremos la segunda opción del menú de diseño.

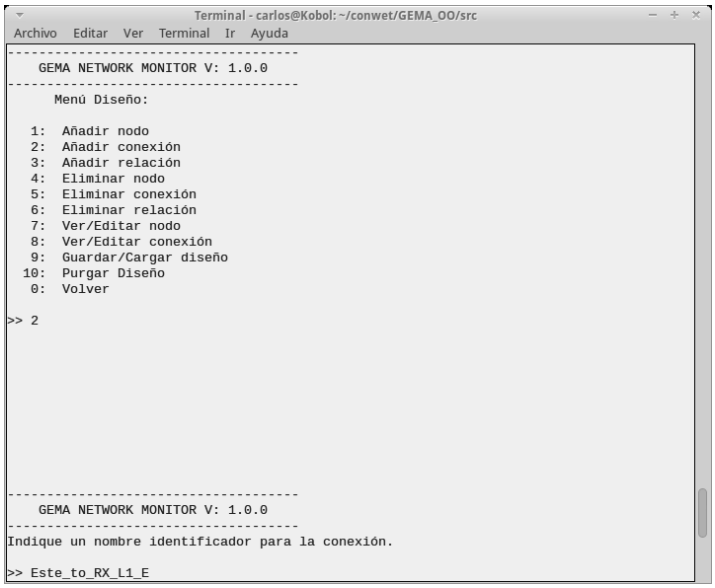


Figura 5.6. Alta de conexiones en GEMA.

El diseño presentado no define ninguna jerarquía de conexiones. Si esta jerarquía estuviese definida se procedería de igual forma que con los nodos. Utilizando la segunda opción del menú de edición de conexiones se mostraría la lista de conexiones disponibles y se solicitaría un alias para cada una seleccionada.

Al finalizar el proceso observamos su implementación en la ventana del monitor.

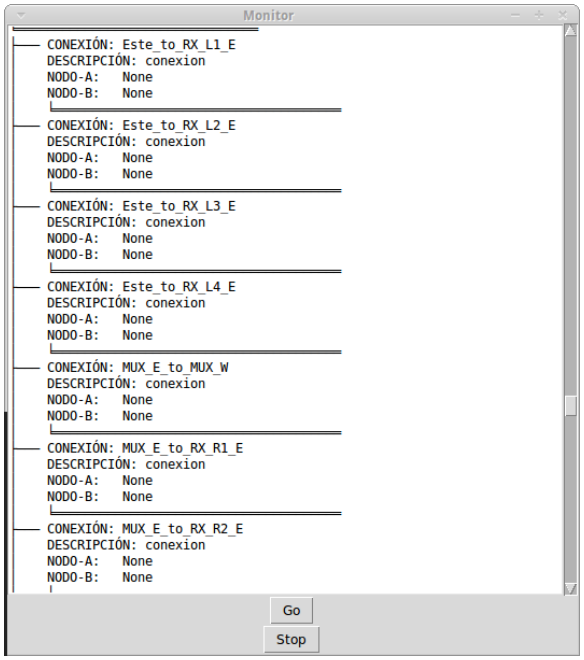


Figura 5.7. Resultado del alta de conexiones en GEMA.

5.2.3. Establecer relaciones

Tras haber dado de alta todos los nodos y conexiones del layout se procede a establecer las relaciones entre ellos. Utilizamos para ello la tercera opción del menú de diseño. Durante el alta de una relación se nos mostrará la lista de nodos disponibles de la que seleccionaremos los dos extremos de la relación, finalmente se seleccionará una conexión de la lista de conexiones disponibles.

Procediendo de la misma manera con las 33 conexiones finalizamos el proceso de alta de la jerarquía del layout en GEMA. El estado final del mismo se podrá comprobar en la ventana de monitorización.

5.3. Declaración de variables

Una vez finalizado el alta de la jerarquía del layout se procede a declarar las diferentes variables que se monitorizarán en cada nodo. En el diseño expuesto solamente existen dos nodos físicos reales, los dos CMUX.

Para dar acceso a GEMA a estos elementos deberemos editar cada nodo y asignarle la dirección IP correspondiente. Asignaremos esta IP a todos los subnodos de cada nodo CMUX. De esta forma todos los componentes de los CMUX representados en el layout por nodos referenciarán al elemento real correspondiente.

A continuación se decide que variables serán monitorizadas en cada dispositivo. Utilizaremos las siguientes variables de cada nodo en este ejemplo:

numberCmux	channelRemoteRx
numberChannel	channelRemoteSfpPresent
cmuxIdentity	channelBitRate
cmuxAlarmStatus	channelLocalByPass
cmuxFanTemp	channelRemoteByPass
cmuxInteriorTemp	channelSDHCompatibility
cmuxVolt3v3	channelSecondary
cmuxVolt2v5	channelActive
cmuxVolt1v2	channelBackup
cmuxLeftPower	channelLoop
cmuxRightPower	channelLocalSfpType
cmuxLeftPowerIdenti	channelLocalSfpConector
cmuxRightPowerIdenti	channelLocalSfpCompatibility
cmuxOpticalSN	channelLocalSfpBitRate
cmuxOpticalPN	channelLocalSfpWaveLength
channelPresent	channelRemoteSfpType
channelLocalTx	channelRemoteSfpConector
channelLocalRx	channelRemoteSfpCompatibility
channelRemote	channelRemoteSfpBitRate
channelLocalSfpPresent	channelRemoteSfpWaveLength

Cada variable de canal indicada representa una variable por cada canal añadiendo el sufijo necesario, por ejemplo *"channelLocalTx.1.1"*

Dado que cada nodo en la jerarquía de un CMUX posee la misma dirección IP podemos declarar las variables en cualquiera de ellos, todas en el nodo raíz del CMUX o cada una en el nodo del layout que le corresponda según el valor que represente. Utilizaremos la segunda opción por ser más representativa del diseño lógico de los CMUX.

Debe tenerse en cuenta que cada nodo que disponga de variables o propiedades creará sus propios threads de actualización. La decisión de agrupar o distribuir estos elementos entre los nodos no es meramente estética o representativa, si no que afectará al nivel de threads activos en el sistema.

Tras asignar las direcciones IP todos los nodos quedan preparados para recibir declaraciones de variables. Para ello, en el menú de edición de cada nodo, seleccionaremos la primera opción.

En primer lugar se nos solicitará un protocolo para utilizar. Cada nodo podrá disponer de diferentes variables utilizando diferentes protocolos. En este ejemplo utilizaremos SNMPv2 para todas ellas.

Figura 5.8. Selección de protocolo en GEMA.

En este punto el protocolo toma el control del interfaz y solicita la información necesaria. En este caso, SNMPv2, solicitará en primer lugar la credencial community que utilizará, esta credencial será la el community por defecto "public"

A continuación se mostrará la lista de MIBs disponibles en GEMA y se solicitará una selección de aquellas que se desean utilizar. En este caso utilizaremos la MIB específica del dispositivo *"CMUX4-PLUS"*.

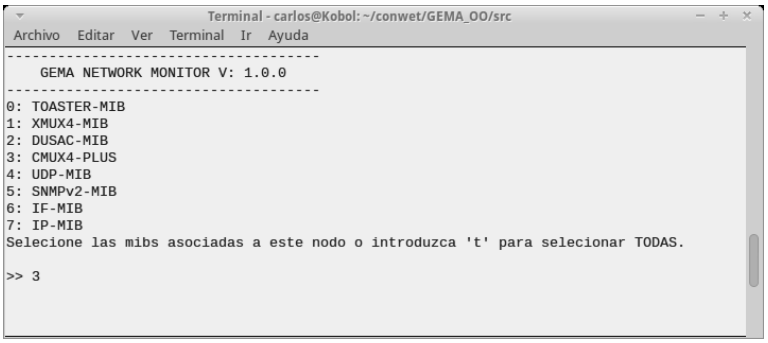


Figura 5.9. Selección de MIBs en SNMPv2.

Inmediatamente el protocolo empezará a comunicarse con el dispositivo utilizando las MIBs seleccionadas. Durante esta comunicación recibirá del dispositivo la lista de variables disponibles descritas por cada una de la MIBs. Esta lista completa se mostrará en pantalla permitiendo al usuario seleccionar las variables deseadas.

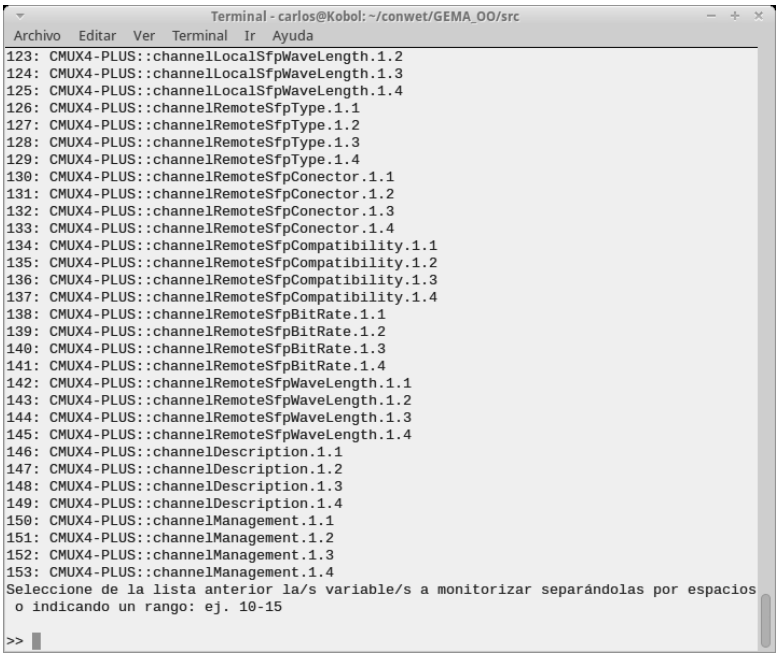


Figura 5.10. Selección de variables en SNMPv2.

La lista de variables seleccionadas queda añadida al nodo correspondiente y se da por finalizada la operación. Se procederá de la misma manera con cada nodo en el layout al que se deseen añadir variables hasta que el diseño quede completo.

De nuevo se ha podido ir observando la evolución del alta de variables en la ventana de monitorización. Tras la adición de cada variable esta ha parecido en la ventana y ha comenzado a actualizar su valor automáticamente.

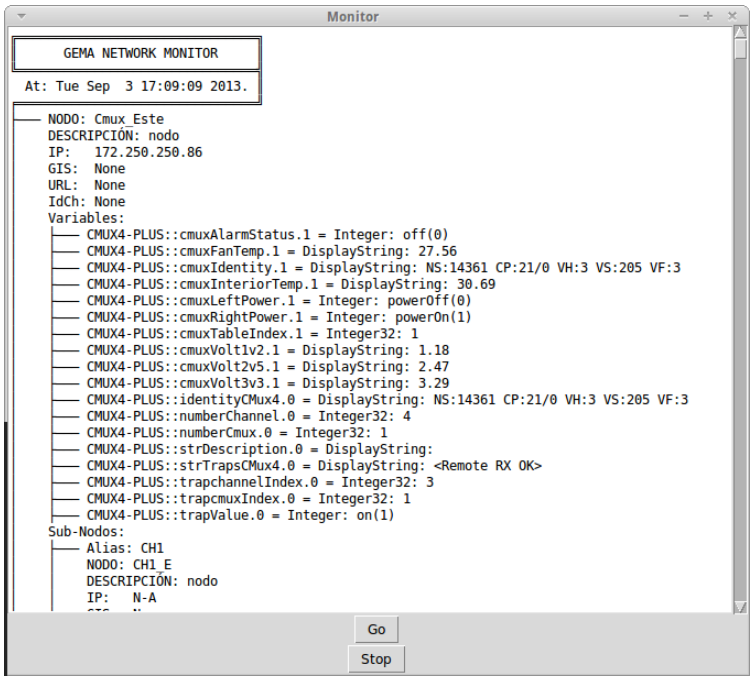


Figura 5.11. Resultado del alta de variables en GEMA.

5.4. Programación de propiedades

En este apartado vamos a añadir algunas propiedades al diseño implementado. Para reducir el nivel de threads en el sistema todas las propiedades se declararán en los nodos raíz de cada CMUX.

En primer lugar se implementarán algunas propiedades síncronas que nos devuelvan el estado general de los nodos en función de los valores particulares de algunas de sus variables. Para ello especificamos las propiedades y definimos el código GEMA-Script de cada una.

Se presentará el código para el CMUX Este, Para el CMUX oeste únicamente se habrá de cambiar el nodo raíz de los direccionamientos.

Potencia del CMUX Ok:

Especificación:	Código:
CmuxLeftPower = 1 *CmuxRightPower=1*	*#%TYPE:Bool* *LeftPower = Cmux_E;FAL;CmuxLeftPower* *RightPower = Cmux_E;FAL;CmuxRightPower* *return LeftPower == 1 && RightPower == 1* *end*

Esta propiedad mostrará un valor "*True*" en caso de que ambas fuentes de alimentación estén presentes y activas.

Warning en Potencia no redundada:

Especificación:	Código:
CmuxLeftPower != CmuxRightPower *CmuxLeftPower == 1 | CmuxRightPower == 1*	*#%TYPE:Bool* *LeftPower = Cmux_E;FAL;CmuxLeftPower* *RightPower = Cmux_E;FAL;CmuxRightPower* *return (LeftPower != RightPower) &&* *(LeftPower ==1 || RightPower ==1)* *end*

Esta propiedad mostrará un valor "*True*" en caso de que únicamente una fuente de alimentación esté presente y activa.

Corte de fibra en dirección E → W:

Especificación:

```
//Los dos multiplexores
      están conectados
CE.cmuxOpticalSN != 0 &
      CW.cmuxOpticalSN != 0

//Al menos un canal está
      transmitiendo y el SFP del
      canal receptor estará
      conectado
(CE.channelRemoteTx.1 = 1 &
CW.channelRemoteSfpPresent.1 = 1) |
(CE.channelRemoteTx.2 = 1 &
CW.channelRemoteSfpPresent.2 = 1) |
(CE.channelRemoteTx.3 = 1 &
CW.channelRemoteSfpPresent.3 = 1) |
(CE.channelRemoteTx.4 = 1 &
CW.channelRemoteSfpPresent.4 = 1)

//Ningún canal está recibiendo
CW.channelRemoteRx.1 = 0 &
CW.channelRemoteRx.2 = 0 &
CW.channelRemoteRx.3 = 0 &
CW.channelRemoteRx.4 = 0
```

Código:

```
#%TYPE:Bool
OpticalEast = Cmux_E;cmuxOpticalSN
OpticalWest = Cmux_O;cmuxOpticalSN
EastC1Tx = Cmux_E;CH1;channelRemoteTx.1
EastC1Tx = Cmux_E;CH2;channelRemoteTx.2
EastC1Tx = Cmux_E;CH3;channelRemoteTx.3
EastC1Tx = Cmux_E;CH4;channelRemoteTx.4
WestC1Rx = Cmux_O;CH1;channelRemoteRx.1
WestC1Rx = Cmux_O;CH2;channelRemoteRx.2
WestC1Rx = Cmux_O;CH3;channelRemoteRx.3
WestC1Rx = Cmux_O;CH4;channelRemoteRx.4
WestC1SFP = Cmux_O;CH1;channelRemoteSfpPresent.1
WestC1SFP = Cmux_O;CH2;channelRemoteSfpPresent.2
WestC1SFP = Cmux_O;CH3;channelRemoteSfpPresent.3
WestC1SFP = Cmux_O;CH4;channelRemoteSfpPresent.4

Cond1 = OpticalEast ! = 0 && OpticalWest != 0
Cond2 = EastC1Tx ==1 && WestC1SFP = 1
Cond3 = EastC2Tx ==1 && WestC2SFP = 1
Cond4 = EastC3Tx ==1 && WestC3SFP = 1
Cond5 = EastC4Tx ==1 && WestC4SFP = 1
Cond6 = WestC1Rx == WestC2Rx == WestC3Rx == WestC4Rx == 0

return Cond1 && (Cond2 || Cond3 || Cond4 || Cond5 ||) && Cond6
end
```

Esta propiedad evalúa variables de ambos dispositivos al tiempo. De esta forma podemos obtener inferencias generales del estado de la red.

Para que esta propiedad devuelva un valor “True” se debe cumplir que ningún canal reciba señal cuando al menos uno de ellos esté transmitiendo y sus correspondientes multiplexores estén conectados.

Se define también una propiedad asíncrona. Esta propiedad generará una notificación de mensaje GEMA en caso de que un SFP sea retirado del dispositivo. Para ello esta propiedad se subscribirá a las notificaciones de Traps.

Su expresión regular filtrará de forma que únicamente atienda a los referentes a una desconexión de SFP. Su valor indicará el “timestamp” de su última activación. El mensaje a generar será el mismo que el contenido en el Trap capturado.

Código:	Expresión regular:
#%TYPE:Timestamp *notify(mensaje,10)* *return time()*	*SFP off*

Para dar de alta estas propiedades en primer lugar accederemos al menú de edición del nodo. Desde este menú crearemos cada una de las propiedades con la segunda opción. Esta opción no consultará el nombre de la propiedad y su tipo. El tipo de la propiedad podrá ser “Sync”, “Traps”, “Logs” o “Gema”. Elegiremos “Sync” para las tres primeras propiedades y “Gema” para la última. De esta forma se indica que la última propiedad es asíncrona y se nos solicitará su expresión regular.

Una vez creadas las propiedades estas aparecerán en la ventana de monitorización con un valor por defecto. Su código aún no ha sido editado y por tanto su valor no es representativo. Para editar cada una de las propiedades utilizaremos la decimoprimera opción.

Esta consultará que propiedad deseamos editar y tras seleccionarla abrirá la venta de edición de las propiedades. Tras editar todas las propiedades podremos ver sus valores calculados en la ventana de monitorización.

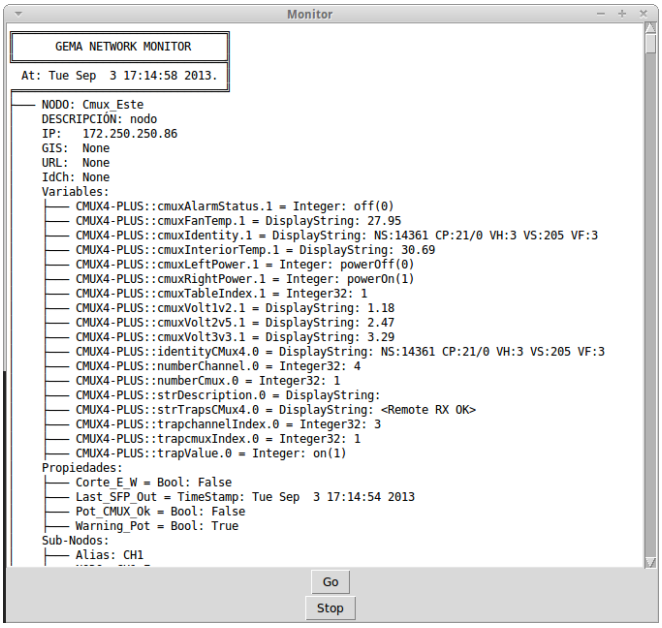


Figura 5.12. Resultado del alta de propiedades en GEMA.

5.5. Configuración de la monitorización

En este punto ya se dispone de un layout completo implementado. Este layout incluye variables y propiedades implementadas. Pero aún es posible configurar algunos parámetros más.

En el menú de edición de nodos y conexiones podemos fijar el valor de algunos de los atributos no activos de estos elementos. Estos atributos, como son la descripción, las coordenadas GIS, la longitud o el sentido en las conexiones no son utilizados directamente por GEMA.

Estos atributos pueden ser accedidos por las propiedades si necesitasen evaluarlos pero están principalmente pensados para su uso por los interfaces gráficos.

Algunos valores, como las coordenadas GIS, podrán ser utilizados para realizar una adecuada representación gráfica o enriquecer la vista de los componentes al usuario.

Desde este menú también podemos configurar que variables y propiedades son actualizadas en cada ciclo e incluso fijar la velocidad de actualización de estos elementos.

En el caso del ejemplo introduciremos una velocidad de actualización de un segundo tanto a variables como a las propiedades en los nodos raíz de cada CMUX. Podremos comprobar que la velocidad de actualización de estos elementos se acelera en la ventana de monitorización.

Una vez realizada esta configuración accederemos al menú de complementos para activar los servicios web, monitor y Api, y lanzaremos el Trapper y el Logger. Hecho esto todos los sistemas de GEMA estarán ahora activos y listos para proceder a la monitorización.

Si en este momento retirásemos un SFP de un multiplexor observaríamos como cambia el valor de la propiedad asíncrona implementada anteriormente.

5.6. Monitorización local

Para la monitorización local disponemos de las ventanas anteriormente descritas. En primer lugar la ventana de monitorización que activamos al inicio del diseño estará actualizándose y mostrando todos los valores de variables y propiedades en el sistema.

Desde el menú de complementos podemos abrir las ventanas de monitorización de Traps y Logs de uno de los nodos. En estas ventanas veremos aparecer en tiempo real todas las entradas que se produzcan debidas a los mensajes capturados por el Trapper y el Logger. Si actuamos sobre el dispositivo observaremos como estas entradas se van actualizando en tiempo real.

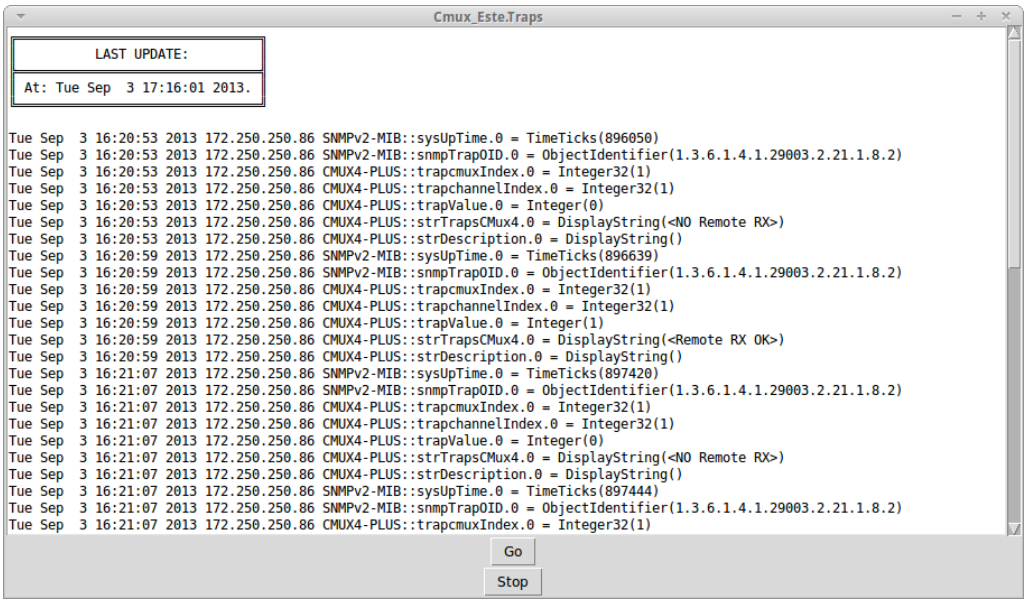


Figura 5.13. Monitorización local de Traps.

Si abrimos una ventana de consola separa podremos observar el estado de los diferentes ficheros de logs. Podremos observar, por ejemplo, la evolución de los logs internos de GEMA haciendo uso del mandato "*tail*".

Con este mandato convertiremos la ventana de consola en un monitor del fichero de logs. Inicialmente mostrará las últimas 100 líneas del fichero. La salida se irá actualizando en tiempo real con cada entrada producida por GEMA en el fichero indicado.

tail –n100 –f /var/log/GEMA/Gema.log

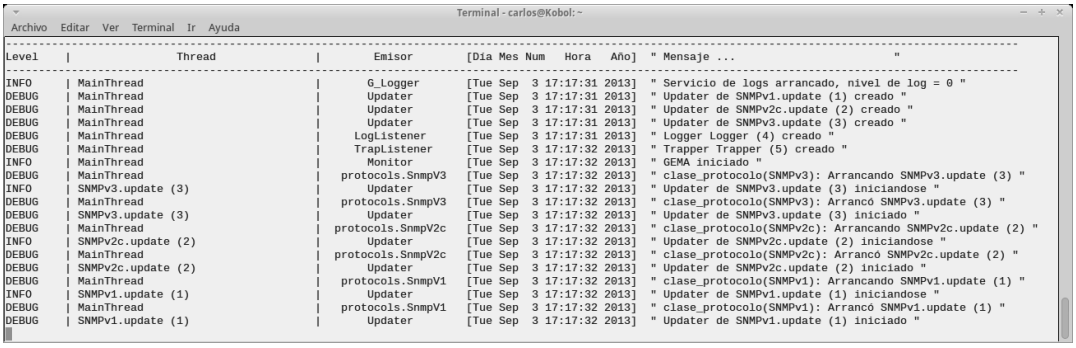


Figura 5.14. Monitorización en consola de Gema.log.

Del mismo modo podremos observar los logs de los servicios web activados.

tail –n100 –f /var/log/GEMA/MonitorWebLog.log

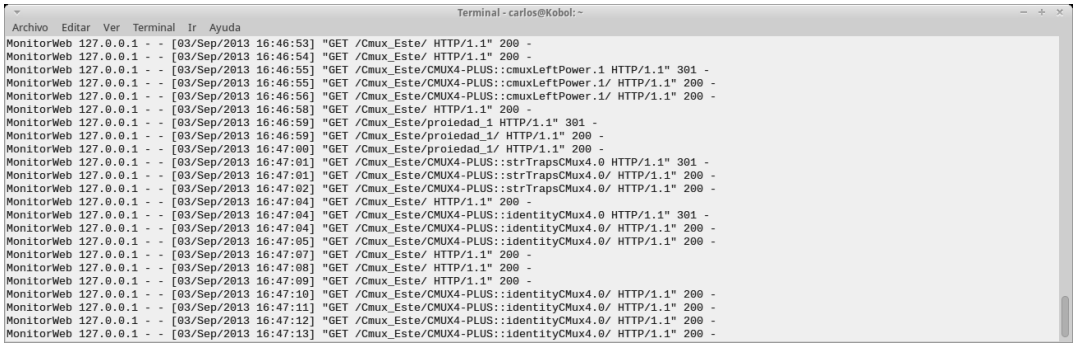


Figura 5.15. Monitorización en consola de MonitorWebLog.log.

tail –n100 –f /var/log/GEMA/ApiWebLog.log

Figura 5.16. Monitorización en consola de ApiWebLog.log.

En estos logs queda registrado cada acceso realizado al servicio web así como los errores que hayan ocurrido.

5.7. Monitorización remota

La monitorización remota es posible gracias a los dos servicios web activados, el monitor web y el Api-Web.

Accederemos a la página web del monitor en la dirección y puertos indicados en el fichero de configuración. Por defecto la página estará disponible en todas las interfaces de la máquina que ejecuta GEMA y el puerto para este servicio será el 8080. De este modo podremos introducir la siguiente dirección el navegador de cualquier dispositivo:

http://mercurio.ls.fi.upm.es:8080

El navegador mostrará la página web con la monitorización actualizada del diseño que hemos introducido.

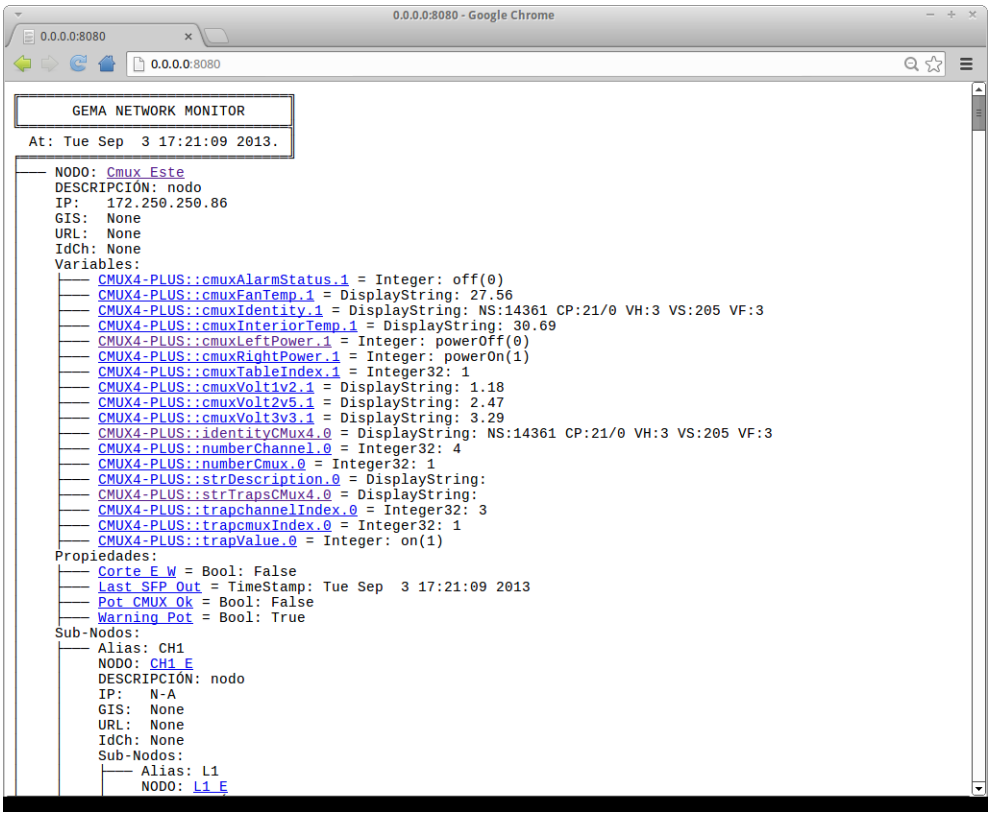


Figura 5.17. Monitorización remota en página web.

También es posible la monitorización mediante una aplicación remota. Por defecto el puerto de escucha del Api-Web es el 8000. Accediendo a esta dirección obtenemos acceso al listado de mandatos soportados por el Api-Web.

Haciendo uso de una aplicación que aproveche estos mandatos en red podemos realizar la monitorización y complementarla con el procesamiento local de los datos monitorizados realizado por la aplicación remota.

Por ejemplo la siguiente aplicación, desarrollada en Java por Fibernet, muestra gráficamente el diseño introducido.

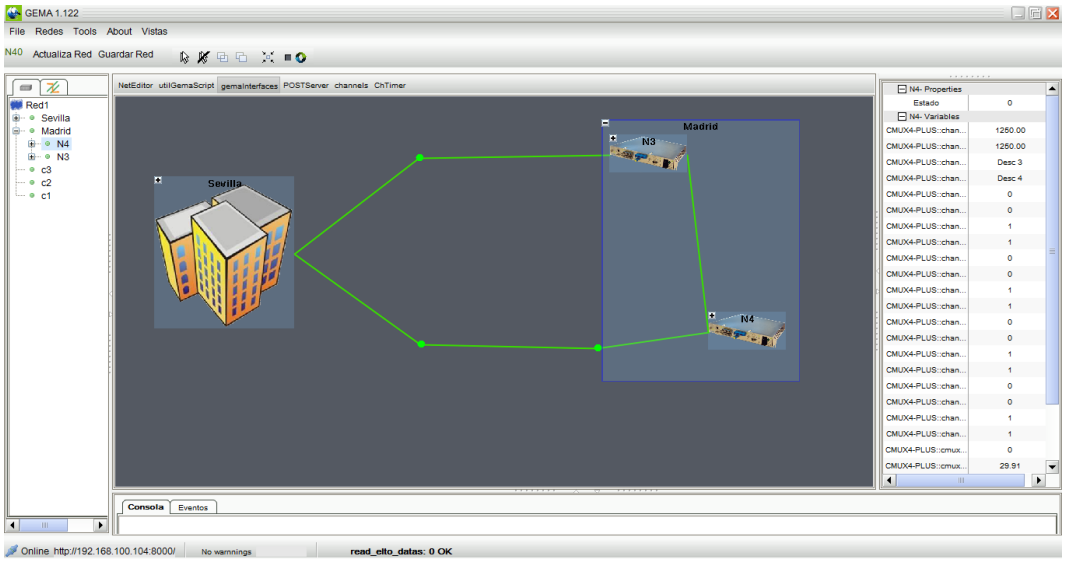


Figura 5.18. Monitorización en aplicación remota.

Haciendo uso de los parámetros no activos de los elementos la aplicación es capaz de representar adecuadamente cada componente. Para ello utiliza diferentes representaciones gráficas en función de estos parámetros.

La aplicación también es capaz, haciendo uso de los recursos de la máquina remota, de realizar cálculos y evaluaciones que GEMA no realice. Un ejemplo de esto es el procesamiento de la información GIS para el posicionamiento sobre un mapa.

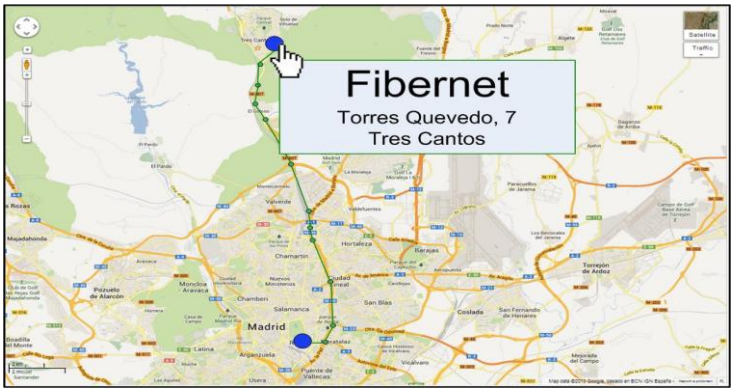


Figura 5.19. Visualización geográfica de nodos.

De esta forma se descarga al servidor de GEMA de los cálculos más pesados y permite que cada aplicación implemente sus propias visualizaciones de los diferentes elementos.

5.8. Escenario de pruebas

Las siguientes imágenes muestran el escenario de pruebas en el que se ha desarrollado el caso de uso expuesto en este capítulo. Para ello se han utilizado dos CMUX4+ sobre los que se realizó la monitorización.

En la primera imagen vemos los dos CMUX utilizados. Estos son los dos representados en el diagrama inicial del caso de uso. Se han realizado las mismas conexiones entre ellos que las indicadas en el diseño.

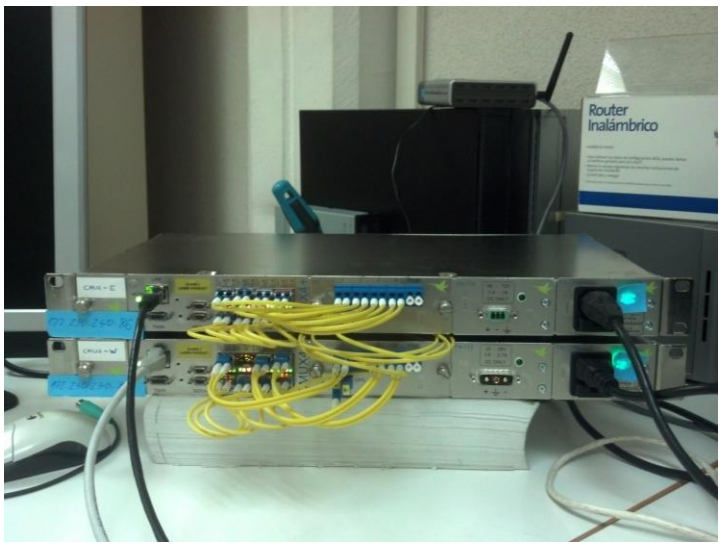


Figura 5.20. Dos CMUX4+ interconectados para simulación del escenario.

Las conexiones internas y entre los dos CMUX, cableado amarillo, son de fibra óptica. Los dos cables de red que se observan son las conexiones de los dispositivos a la red de monitorización.

También se puede observar que de las dos fuentes de alimentación disponibles únicamente se está utilizando la de 220V con un conector de ordenador estándar.

En la siguiente imagen se observa un detalle del conexionado con fibra óptica entre los dos CMUX.

Para simular la conexión a las nubes se utilizan conexiones en bucle entre los canales de transmisión y recepción indicados.

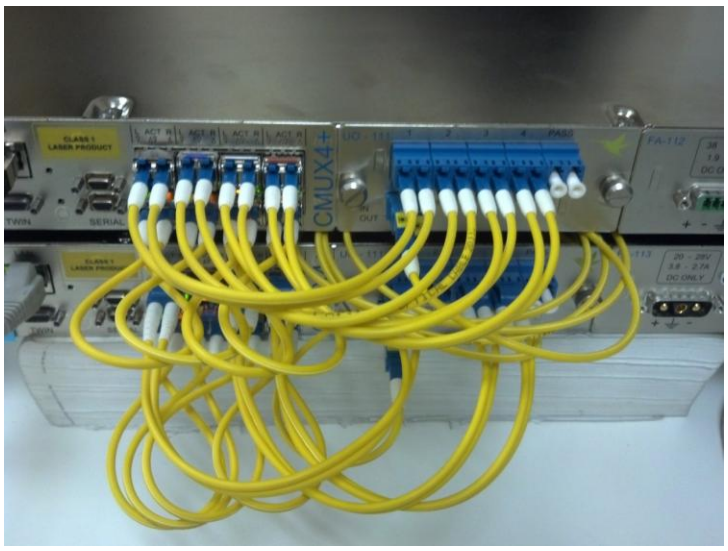


Figura 5.21. Detalle del conexionado de los CMUX4+.

Las conexiones del bloque izquierdo al derecho corresponden a las conexiones entre los SFP, a la izquierda, y el multiplexor, modulo derecho.

Pueden observarse los lazos de las conexiones en bucle simulando la nube en la segunda hilera del primer CMUX.

El conexionado entre los dos dispositivos se realiza mediante una única fibra larga que se puede observar enrollada entre los dos CMUX.

Para la ejecución de GEMA se ha utilizado un Core 2 Quad a 2.33Ghz con 4GB de RAM sin que haya sufrido ninguna sobrecarga con todos los servicios activados y varias ventanas de monitorización, tanto de GEMA como en consola, abiertas al mismo tiempo.

Durante las pruebas se han ejecutado varios software simulando aplicaciones remotas en diferentes equipos de la red tratando de saturar las peticiones al API-Web. También se han realizado accesos simultáneos por medio de navegadores web con refresco a cada segundo.

Durante todas las pruebas GEMA se ha mostrado estable y sin síntomas de sobrecarga en el sistema incluso durante largos periodos de funcionamiento.

Capítulo 6

Conclusiones y Líneas futuras

Índice

En este proyecto se ha llevado a cabo la realización de una aplicación de gestión y monitorización de redes. Se ha realizado un análisis de las técnicas y herramientas actuales, se ha expuesto el diseño realizado, las metodologías y las tecnologías utilizadas.

Se ha realizado una descripción modulo a modulo del mismo y posteriormente se ha presentado el resultado final del desarrollo realizado en una breve exposición de la aplicación y un caso de uso de pruebas. Finalmente, en este capítulo, se presentan una serie de conclusiones, tanto técnicas como personales, que se han alcanzado durante la realización de este proyecto.

Tras estas conclusiones se exponen las posibles líneas futuras de desarrollo del proyecto, tanto modificaciones y mejoras sobre las funcionalidades actuales como la adición de nuevos módulos que doten a la aplicación de una mayor capacidad de trabajo en el entrono de la monitorización y gestión de redes.

6.1. Conclusiones técnicas

Al comienzo de este texto se analizaron diversas alternativas software de gestión y monitorización de redes. En base a ese estudio y a las reuniones con los representantes de Fibernet se determinaron los diferentes objetivos que debería cumplir la aplicación a su finalización. En este punto, con una versión completa de la misma, se puede decir que esos objetivos se han satisfecho adecuadamente.

Se ha desarrollado una aplicación funcional de monitorización y gestión de redes multiprotocolo con amplia capacidad de evolución y adaptación a las nuevas tecnologías.

El diseño modular, orientado a objetos, utilizado la dota de una gran capacidad de expansión y personalización, pudiendo añadirse o modificarse cualquier módulo sin propagar excesivos cambios al resto de elementos que la componen. Este diseño ha permitido la incorporación de módulos de forma independiente implementados con sus propios microframeworks como es el caso de Flask, utilizado para implementar los servicios web.

Es capaz de monitorizar cualquier tipo de red y permite definir en detalle todos los elementos desde cero. Cuenta con la capacidad de describir tanto física como lógicamente la red y los elementos gestionados. El modelo de diseño utilizado para definir las redes le otorga una amplia versatilidad a la hora de describir las diferentes topologías y tecnologías de red actuales y futuras.

La capacidad de programación de la que se le ha dotado le otorga una potencial capacidad de tratamiento de diferentes situaciones y de análisis de datos más allá de la que se esperaba al comienzo del proyecto. El lenguaje GEMA-Script puede ser utilizado para diversas operaciones desde simples verificaciones lógicas hasta comportamientos automáticos basados en diferentes técnicas de inferencia sobre los datos obtenidos.

Permite el acceso remoto por diferentes medios, permitiendo utilizar diferentes interfaces con el usuario a distintos niveles. Los accesos se podrán realizar desde simples navegadores web a los servicios que se habiliten desde el servidor o por aplicaciones complejas que conectan con la aplicación mediante el api-web desarrollado.

Tras la finalización del proyecto se considera que las elecciones de tecnología, paradigma de orientación a objetos, metodología de prototipado rápido en ciclo de desarrollo espiral, lenguaje de programación Python, etc. han resultado muy convenientes pues han generado más beneficios que perjuicios.

Algunos de los perjuicios encontrados han sido, en la parte teórica, la necesidad de documentarse y aprender técnicas específicas del lenguaje o diseño de patrones del paradigma que se adecuasen al proyecto. En el ámbito del desarrollo del proyecto la metodología elegida obligaba a cumplimiento de plazos en la entrega y presentación de prototipos y obligaba a desviar parte del tiempo de trabajo a la preparación de estos del desarrollo del proyecto.

Sin embargo estos perjuicios se han visto superados ampliamente por la flexibilidad ante cambios y nuevos requisitos en el proyecto. Los cuales han ido apareciendo bastante a menudo, tanto por confusiones en la comprensión de algunos de ellos como por nuevas funcionalidades requeridas por parte de Fibernet.

La versatilidad de Python y su sencillez de programación han permitido centrar gran parte del esfuerzo en el diseño y el análisis de las funcionalidades de la aplicación. Permitiendo que sea el lenguaje y el código los que se adecuen al problema y no al contrario sin tener que realizar un gran esfuerzo para ello.

Como conclusión final a este apartado indicar que tanto el equipo de desarrollo del proyecto GEMA del CoNWeT Lab como el equipo de Fibernet implicado han quedado ampliamente satisfechos con el proyecto realizado con una valoración muy positiva del producto obtenido.

6.2. Conclusiones Personales

Tras haber concluido el proyecto GEMA tras un año de trabajo en el CoNWeT Lab y en colaboración con Fibernet mi valoración general es muy positiva en todos los aspectos.

A nivel académico este proyecto me ha ayudado a adquirir nuevos conocimientos en tecnologías de redes, como SNMP o HTTP. También he incrementado ampliamente mis conocimientos de programación en Python y la puesta en práctica de muchos de los conocimientos teóricos recibidos durante la carrera ha ayudado a afianzarlos y comprenderlos mejor.

Desde un punto de vista laboral, el trabajar en un equipo completo, formado tanto por alumnos becados en el CoNWeT Lab, personal de la universidad y empleados de Fibernet me ha llevado a desempeñar un estilo de trabajo profesional en el que se han de cumplir plazos y objetivos y en el que se ha de realizar un trabajo en equipo en el que las diversas tareas se reparten y se debe cumplir el compromiso de cada uno con el resto de miembros del equipo.

La dinámica de trabajo, basada en el modelo de prototipado expuesto, ha resultado muy satisfactoria. La presentación de sucesivos prototipos y la recepción de los comentarios, sugerencias y opiniones por parte del cliente resulta un incentivo a la hora de realizar el trabajo cuando son positivas y un revulsivo cuando no lo son, haciendo que se maximicen los esfuerzos de cara a la siguiente presentación.

Las reuniones internas semanales de los miembros del equipo, becarios y tutores, ayudan a mantener un entorno de trabajo unificado y a poner todas las ideas en común de cara a aunar los esfuerzos y dirección del proyecto. De esta forma el producto final es un todo cohesionado en lugar de un agregado de diferentes componentes sin una visión común

Esta existencia de plazos e hitos a cumplir también me ha enseñado que eventualmente se debe dar un trabajo por realizado, no insistiendo en una optimización teórica máxima que raramente permite cumplir las fechas de entrega o alcanzar los hitos marcados.

La necesidad de enviar informes semanales y mantener una comunicación constante con los encargados de Fibernet me ha mostrado el entorno de trabajo profesional en una empresa. Un entrono en el que hay diferentes cargos y responsabilidades en el que cada uno debe atenerse a su nivel y las competencias que ello le otorga, tanto en decisiones de diseño como en establecimiento de plazos e hitos de un proyecto.

Desde un punto de vista más personal decir que he quedado enormemente satisfecho del entorno y ambiente de trabajo tanto con todos los compañeros y personal del CoNWeT Lab y la universidad como con el personal de Fibernet. En todo momento ha sido un ambiente cordial de ayuda mutua y colaboración que desearía volver a encontrar en mi futuro profesional.

6.3. Líneas Futuras

En todo proyecto se han de cumplir unos plazos y unos objetivos marcados. Estos plazos muchas veces impiden desarrollar al máximo todas las posibles funcionalidades que el proyecto podría tener u optimizar todas las implementadas al 100%. Este proyecto no es una excepción, a la finalización de esta versión, GEMAv1, se han alcanzado todos los hitos propuestos para la misma y el nivel de satisfacción general con respecto al grado de desarrollo alcanzado es muy alto. Sin embargo quedan funcionalidades que podrían añadírsele y otras de las que dispone podrían ser mejoradas.

Algunas de estas funcionalidades se detectaron durante la fase de estudio del estado del arte y otras aparecieron como resultado inevitable del desarrollo del diseño e implementación de la aplicación. Es inevitable, cuando se están diseñando y desarrollando funcionalidades o depurando las ya existentes, que aparezca la pregunta "¿Y si…? O que durante las pruebas del producto obtenido surjan casos particulares o carencias en ciertos aspectos.

Si se hubiera pretendido fijar todas estas ideas en una primera etapa de desarrollo del proyecto posiblemente haría no mucho que hubiera salido de la etapa de captura de requisitos y se estarían aún programando las capas más internas del núcleo de la aplicación. Para que, inevitablemente, acabasen surgiendo dudas, discrepancias o carencias no tenidas en cuenta.

Incluso el propio avance tecnológico en el campo de redes de computadores es una fuente de incertidumbre respecto a los requisitos que se exigirán al proyecto una vez acabado que no es posible tener en cuenta al 100% durante las primeras etapas de diseño.

Algunas de las funcionalidades, o líneas de trabajo futuras de que se podría dotar a GEMA para su siguiente versión son las que siguen:

Añadir seguridad a las conexiones al Api-Web y al monitor Web por SSL. En el estado actual de las redes de telecomunicaciones y más en el campo de gestión de dispositivos de administración de redes es prioritario asegurar la seguridad de las conexiones y la autenticidad de las órdenes recibidas por GEMA desde el exterior.

Añadir un módulo de gestión de usuarios que permita identificar las operaciones realizadas y administrar algún mecanismo de niveles de permisos dentro de GEMA. Un modulo de estas características permitiría a GEMA gestionar varias redes independientes con diferentes grupos de usuarios, monitores y administradores. Cada uno de estos grupos podría tener sus propios privilegios de gestión y monitorización específicos.

Como se comentó en el capítulo 3, el intérprete estándar de Python, CPython, consta de un GIL, el cual impide la paralelización de threads. Existen otros intérpretes y técnicas específicas de programación en Python para dotarle de paralelización. Analizar estas opciones e implementar mecanismos que permitan la paralelización de GEMA sería un enorme avance en su optimización y aprovechamiento de recursos hardware. Actualmente cualquier ordenador personal de uso doméstico ya cuenta como mínimo con procesadores de doble núcleo.

Los interfaces gráficos propios de GEMA, ventanas locales y página web de monitorización pueden ser ampliamente mejorados. Estos interfaces requerirían de amplias mejoras tanto en calidad de presentación como de funcionalidad y usabilidad. La enorme variedad de frameworks de desarrollo de interfaces gráficas hace esta una más que viable mejora de la aplicación.

Perfeccionar GEMA-Script. El lenguaje de scripting de GEMA actual es completo y sencillo, pero carece de ciertos recursos o azucares sintácticos que faciliten una cómoda programación de diseños complejos. El desarrollo de un nuevo lenguaje o la mejora del actual añadiendo este tipo de recursos podría facilitar enormemente el desarrollo de scripts complejos que realicen tareas automáticas y tomen decisiones basados en diferentes técnicas de inferencia avanzadas.

Otro punto de mejora, aunque externo en parte a GEMA, es el desarrollar scripts avanzados. Dotar a GEMA de un catálogo de scripts avanzados que implementen mecanismos de inteligencia artificial para la diagnosis de redes y actuación automática sería un valor añadido muy a tener en cuenta. Sobre todo de cara a una posible comercialización de producto.

Al igual que el punto anterior, un gran valor añadido sería la inclusión de módulos análisis de datos. Módulos de análisis de datos avanzados, que implementen técnicas de data-mining o de diagnostico del estado de la red serían bien recibidos. Igualmente módulos de almacenamiento y acceso a información basados en bases de datos, como una base de datos round robin para los valores monitorizados facilitaría enormemente las tareas de diagnostico, análisis y seguimiento de las redes.

Finalmente, algún módulo que permita cierto nivel de compatibilidad con los plugins de otros software de gestión de redes, como Nagios o Zennos, sería un paso adelante en la implantación y compatibilidad del proyecto dentro del mercado.

Esta lista de posibles líneas futuras no pretende ser ni mucho menos exhaustiva. Seguramente queden muchísimas más líneas de mejora y funcionalidades que se podrían añadir a un proyecto de estas características. El campo de la gestión, monitorización y administración de redes es enorme y las tecnologías para automatizar estas labores también lo son.

Con esta última línea concluyo el trabajo de doce meses en este proyecto satisfecho de mi participación y aportación al mismo, esperando esa misma satisfacción por parte del resto de integrantes del proyecto GEMA junto a los que ha sido un placer trabajar.

Referencias y Bibliografía

[1] J. Case et al. *A Simple Network Management Protocol (SNMP)*, IETF RFC 1157, May 1990; http://tools.ietf.org/html/rfc1157, [Sep. 3, 2013].

[2] J. Davin et al. *A Simple Gateway Monitoring Protocol*, IETF RFC 1028, Nov 1987; http://tools.ietf.org/html/rfc1028, [Sep. 3, 2013].

[3] S. Waldbusser. *Remote Network Monitoring Management Information Base*, IETF RFC 2819, May 2000; http://tools.ietf.org/html/rfc2819, [Sep. 3, 2013].

[4] U. Warrier et al. *The Common Management Information Services and Protocols for the Internet (CMOT and CMIP)*, IETF RFC 1189, Oct 1990; http://tools.ietf.org/html/rfc1189, [Sep. 3, 2013].

[5] D. Harrington, R Presuhn and B. Wijnen. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*, IETF RFC 3411, Dec 2002; http://tools.ietf.org/html/rfc3411, [Sep. 3, 2013].

[6] William Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*, 3rd ed. MA: Addison Wesley, 1999.

[7] William Stallings. *SNMP, SNMP2 and CIMP: The practical guide to network management standards.* MA: Addison Wesley, 1999.

[8] *A Brief Tour of the Simple Network Management Protocol*. CERT® Coordination Center, 2002.

[9] S. Legg. *Encoding Instructions for the Generic String Encoding Rules (GSER)*, IETF RFC 4792, Jan 2007; http://tools.ietf.org/html/rfc4792, [Sep. 3, 2013].

[10] Douglas Mauro and Kevin Schmidt. *Essential SNMP*. CA: O´RELLY, 2001.

[11] *Data Encryption Standard (DES)*, FIPS 46-3, Oct 1999; http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, [Sep. 3, 2013].

[12] S. Turner and L. Chen. *Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*, IETF RFC 6151, Mar 2011; http://tools.ietf.org/html/rfc6151, [Sep. 3, 2013].

[13] *Secure Hash Standard (SHS)*, FIPS 180-4, Mar 2012; http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf, [Sep. 3, 2013].

[14] Jeffrey D. Case et al. *Introduction to Version 3 of the Internet-standard Network Management Framework*, IETF Internet draft, Jun 1998; http://tools.ietf.org/html/draft-ietf-snmpv3-intro-00, [Sep. 3, 2013].

[15] "Nagios"; http://www.nagios.org, [Sep. 3, 2013].

[16] "Zennos Community"; http://community.zenoss.org, [Sep. 3, 2013].

[17] "RRDTool"; http://oss.oetiker.ch/rrdtool/, [Sep. 3, 2013].

[18] "Castle Rock Computing"; http://www.castlerock.com, [Sep. 3, 2013].

[19] "Cisco Prime for IP Next Generation Networks"; http://www.cisco.com/en/US/products/ps12290/index.html, [Sep. 3, 2013].

[20] "Zabbix"; http://www.zabbix.com/es/, [Sep. 3, 2013].

[21] "Cacti"; http://www.cacti.net/, [Sep. 3, 2013].

[22] "Centreon"; http://www.centreon.com/, [Sep. 3, 2013].

[23] "Extensible Markup Language"; http://www.w3.org/xml/, [Sep. 3, 2013].

[24] "Python Programming Language"; http://www.python.org/, [Sep. 3, 2013].

[25] "The PySNMP Project"; http://pysnmp.sourceforge.net/, [Sep. 3, 2013].

[26] "The Eric Python IDE"; http://eric-ide.python-projects.org/, [Sep. 3, 2013].

[27] "Subversion"; http://subversion.apache.org/, [Sep. 3, 2013].

[28] "TkInter"; http://wiki.python.org/moin/TkInter, [Sep. 3, 2013].

[29] "Flask"; http://flask.pocoo.org/, [Sep. 3, 2013].

[30] "HyperText Markup Languaje"; http://www.w3.org/html/, [Sep. 3, 2013].

[31] Raúl González Duque. *Python para todos.* [Pdf]. Available: http://mundogeek.net/tutorial-python/ [Sep. 3, 2013].

[32] "Plex"; http://pythonhosted.org/plex/, [Sep. 3, 2013].

[33] Fernando Alonso, Loïc Martínez and Fco. Javier Segovia. *Introducción a la ingeniería del software: Modelos de desarrollo de programas.* Delta Publicaciones, 2005.

[34] Mark Lutz and David Ascher. *Learning Python,* 2nd ed. CA: O´RELLY, 2003.

Glosario y Acrónimos

A

AFD:	Autómata finito determinista.
Alias:	Identificador alternativo de un elemento dentro de un determinado ámbito.
API:	Application Programming Interface.
Asíncrono:	Evento cuya ocurrencia no está regulada por un reloj o sistema de control y cuya frecuencia no puede ser determinada.
ASN1:	Abstract Syntax Notation One.

C

Callback:	Función a ejecutar tras la ocurrencia de un evento determinado. Normalmente esta función es parámetro del método de detección de eventos.
CMIP:	Common Management Information Protocol.
Conexión:	Elemento virtual del layout que representa un enlace entre dispositivos de la red.
Consola:	Interfaz de línea de comandos del sistema operativo o de una aplicación.
CRUD:	Create, Read, Update & Delete.

D

DES:	Data Encryption Standard.
Diccionario:	Un diccionario Python es una estructura de datos en la cual se asocia explícitamente un valor de un conjunto de índices (denominados claves) a un valor de otro conjunto de datos (llamado valor) constituyéndose un conjunto de pares (clave, valor).
Dirección IP:	Etiqueta numérica que identifica, de manera lógica y jerárquica, a un interfaz de un dispositivo dentro de una red que utilice el protocolo IP.

Directiva: Orden dirigida a un compilador dentro del código de un programa para indicar una operación o mecánica específica durante la compilación.

DNS: Domain Name System.

DWDM: Dense Wavelength-Division Multiplexing.

E

Expresión regular: Secuencia de caracteres que define un patrón.

F

Fibra óptica: Medio de transmisión empleado habitualmente en redes de datos. De compone de un hilo muy fino de material transparente, vidrio o materiales plásticos, por el que se envían pulsos de luz que representan los datos a transmitir.

Flag: Variable utilizada para representar el estado de un proceso o elemento del programa.

Front-End: Porción de una aplicación que interactúa con el usuario.

FTP: File Transfer Protocol.

G

GEMA: Gestor Multinodo Avanzado.

GIL: Global interpreter lock.

GIS: Geographic Information System.

GPL: General Public License.

GUI: Graphical User Interface.

H

HMAC: Hash Message Authentication Code.

HTTP: Hypertext Transfer Protocol.

I

ICMP: Internet Control Message Protocol.

ICMP: Internet Control Message Protocol.

IETF: Internet Engineering Task Force.

Import: Operación o directiva de un lenguaje de programación para solicitar la carga de un módulo.

IP: Internet Protocol.

IPMI: Intelligent Plataform Management Interface.

L

Layout: Conjunto organizado de nodos, conexiones, variables, propiedades y protocolos que definen una red a monitorizar.

LDAP: Lightweight Directory Access Protocol.

Léxico: Conjunto de palabras que conforman un determinado lenguaje. En un compilador estas palabras podrán venir definidas mediante su enumeración o mediante su descripción utilizando expresiones regulares.

LL: Left to Right Leftmost derivation.

Log: Entrada o conjunto de entradas de notificación de sucesos de un sistema.

M

MAP: Función de orden superior que aplica la función dada a una lista de elementos devolviendo la lista de resultados.

MD5: Message-Digest Algorithm.

MIB: Management Information Base.

Multiplexor: Dispositivo que puede recibir varias entradas y transmitirlas por un medio de transmisión compartido.

N

NNTP: Network News Transport Protocol.

Nodo: Elemento virtual del layout que representa un dispositivo monitorizable de la red.

O

OID: Object Identifier.

P

Paradigma: Conjunto de definiciones, estrategias y tecnologías utilizadas para el diseño y desarrollo de software.

Plugin: Complemento a una aplicación.

Poliárbol: Grafo dirigido acíclico que puede constar con más de un nodo origen.

POP: Post Office Protocol.

Proceso: Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados.

Producción: Regla que define la expansión o derivación de un símbolo no terminal en una gramática.

Propiedad: Función monitorizable con capacidad de lectura y actuación sobre la aplicación y los dispositivos

Protocolo: Conjunto de reglas y estándares que controlan la secuencia de mensajes que ocurren durante una comunicación entre entidades que forman una red.

R

RFC: Request For Comments.

RMON: Remote Network Monitoring.

S

Script: Conjunto de declaraciones y ordenes secuenciales que implementan una propiedad.

SDK: Software Development Kit.

Serialización: Proceso de aplanado de un conjunto de datos para su almacenamiento o transmisión.

SFP: Small form-factor pluggable transceptor.

SGMP: Simple Gateway Monitoring Protocol.

SHA: Secure Hash Algorithm.

Síncrono: Evento cuya ocurrencia está regulada por un reloj o sistema de control y dispone de una frecuencia determinada.

Singleton: Patrón de diseño caracterizado por restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Sintáctico:	Conjunto de reglas que definen la formación de estructuras dentro de un lenguaje.
SMI:	Structure of Management Information.
SMP:	Simple Management Protocol.
SMS:	Short Message Service.
SMTP:	Simple Mail Transfer Protocol.
SNMP:	Simple Network Management Protocol.
Socket:	Proceso o hilo existente en la máquina cliente y en la máquina servidora, que sirve, en última instancia, para que el programa servidor y el cliente lean y escriban la información. Esta información será la transmitida por las diferentes capas de red.
SONET/SDH:	Synchronous Optical Network's Synchronous Digital Hierarchy.
SQL:	Structured Query Language.
SSH:	Secure Shell.
SSL:	Secure Sockets Layer.

T

TCP:	Transmission Control Protocol.
TDM:	Time Division Multiplexing.
Término:	Significado
Thread:	Unidad de procesamiento más pequeña que puede ser planificada por el sistema operativo.
Timestamp:	Secuencia de caracteres que denotan la hora y fecha en la cual ocurrió determinado evento.
Token:	Elemento básico de un lenguaje de programación representativo de una palabra del léxico del mismo.
Trap:	Mensaje de notificación asíncrono del protocolo SNMP.

U

UDP:	User Datagram Protocol.
UML:	Unified Modeling Language.
USM:	User-based Security Model.

V

VACM:	View-based Access Control Model.
Variable:	Valor monitorizable de un nodo.
VPN:	Virtual Private Network.

W

WSON:	Wavelength Switched Optical Networks.

X

XML:	Extensible Markup Language.